Amiga® Mail

Volume II

November/December 1991

The Commodore® Amiga Technical Newsletter

Copyright © 1991 by Commodore-Amiga, Inc, 1200 Wilson Drive, West Chester, PA, 19380. All rights reserved. Printed in the USA. Authors' materials used herein have been reproduced with the permission of the authors indicated. Use or reproduction of such materials shall be in accordance with the authors' instructions. Commodore and CDTV are registered trademarks of Commodore Electronics Ltd. Amiga is a registered trademark of Commodore-Amiga, Inc. Kickstart, Workbench, AmigaDOS, Amiga 500, Amiga 2000, and Amiga 3000 are trademarks of Commodore-Amiga, Inc. Unix is a registered trademark of AT&T Bell Laboratories, Inc. This document may also contain reference to other trademarks and registered trademarks for the various products listed which are believed to belong to the sources associated therewith.

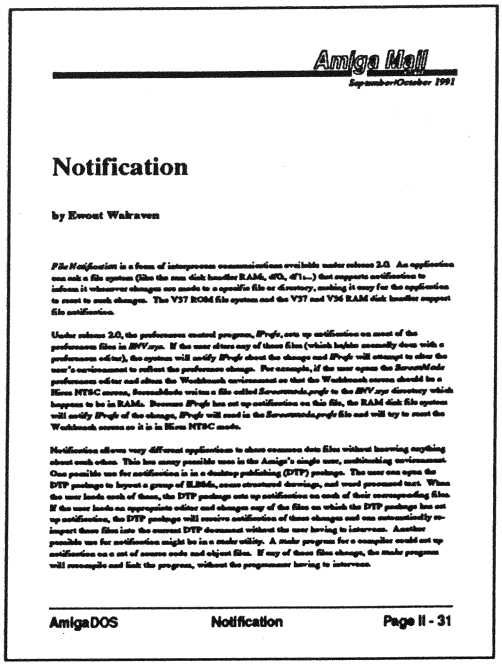
Amiga Mail
Volume II

Amiga Mail is published six times annually by the Commodore Applications and Technical Support Group (CATS). Its purpose is to support third-party developers of Amiga software and hardware worldwide by providing technical information and programming guidelines. Each issue features a collection of news and technical articles of special interest to Amiga product developers.

The Amiga Mail Reference Book (Volume II)

Collectively, the articles from this issue and past issues of Amiga Mail form a reference book. The book is broken up into chapters covering different aspects of Amiga programming (Exec, IFF, AmigaDOS...). All Amiga Mail issues are printed on three hole punch paper so they can easily be filed in a three ring reference book binder. To file the articles from this issue in the reference book, use the chapter and page numbers that appear at the foot of each page. Each page's chapter number is shown as a Roman numeral followed by its page number within that chapter.

For instance, the Amiga Mail article below "Notification", would be filed in chapter II of the binder, which is the "AmigaDOS" chapter. The article starts on page 31 of that chapter in the binder.


Amiga Mail
September/October 1991

Notification

by Ewout Wakraven

AmigaDOS Notification Page II - 31

Each issue of Amiga Mail also comes with an update to the book's table of contents.

From time to time, an issue of Amiga Mail is published with correction pages which replace existing pages in the Amiga Mail reference book. These appear at the end of an Amiga Mail issue after the Table of Contents update.

For more information on how to obtain a binder, chapter separators, and past issues of Amiga Mail, contact your local Commodore support manager, or, if you are in the U.S., contact CATS at (215) 431-9180.

Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

Disclaimer

This information is provided "As Is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any claim arising out of the information presented herein, even if it has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of such implied warranties, so the above limitations may not apply.

Amiga Mail Editor:
John Orr

Contributors:
Susan West, Michael Sinz, Carolyn Scheppner, David Junod, Randell Jesup, Brian Jackson, Darren Greenwald, Peter Cherna and Dan Baker.

About Amiga Mail:
Amiga Mail is published on an Amiga 3000.
The software used to produce this issue includes: *PageStream 2.1*, New Horizon's *ProWrite*, the *SAS/C Amiga C Compiler*, Oxxi's *TurboText*, and Softwood, Inc.'s *Proper Grammar*.

Comments to:

John Orr
CATS
Commodore Business Machines, Inc.
1200 Wilson Drive
West Chester, PA 19380

or

via E-mail: johno@cbmvax.commodore.com
[rutgers| uunet] ! cbmvax ! johno

This publication is printed on recycled material.

Amiga Mail News

1991 Amiga DevCons

Denver and Milan were the sites of Amiga developer conferences during the first two weeks of September. This is the first year the conferences have been held back-to-back, with the objective providing the most current information to the entire developer community.

The North American conference drew 265 attendees from six countries (U.S.A., Canada, Germany, France, Austria, and Australia). DevCon Milano was attended by 125 Amiga and CDTV developers. This year's DevCon Notes contain over 800 pages of documentation covering 33 sessions including Intro to Amiga Programming, New High-End Hardware, Localization and CDTV topics. (DevCon Notes are now available from CATS-Orders for $125.00 a set.)

Merit of the conference overall was rated Excellent or Very Good by 86% of attendees. Feedback forms indicate that attendees continue to rate "Networking Value" as the highest attribute of the conference, with 42% giving it an Excellent Rating, 37% rating it Very Good. Attendees had several good suggestions for improving DevCon, including the addition of special-interest round table discussions to identify developers with the same interests to further improve the networking value of the conference.

Some DevCon highlights:

Keynote Address

Jeff Scherb, Vice President, CATS, presented the keynote address, which outlined the accomplishments of Commodore and the development community over the past year. The following is excerpted from his address:

- We are now getting the press coverage we deserve. The Amiga is finally being recognized as a powerful and cost-effective alternative to the PC and Macintosh. We are getting noticed by the professional computing community.
- The A3000 is shipping in volume. The recent "Power Up" sales promotion run by Commodore in the US has exceeded all of our expectations, and has, in fact, resulted in a backlog of orders for the machine.
- The A500 continues to gain momentum, particularly in Europe. Many believe that the 500 is now poised to repeat the success of the Commodore 64.
- CDTV is shipping, to tremendous press reviews. Over 100 titles are already available for the machine. The press now defines the "interactive multimedia player" machine generically as CDTV, rather than CD-I as it did a year ago. We are now the leader that the "other guys" have to follow.
- We are now shipping Amiga running AT&T System V, Release 4 UNIX. This is one of the earliest commercial implementations of V.4, and our implementation includes *Open Look* and color *X Windows* running on the University of Lowell graphics card. Over 150 applications are already available for the machine.
- The A3000T "tower" machine is now shipping -- this elevates Amiga expandability and performance to new levels.
- AmigaDOS 2.0 is now finished, and ROMs are being manufactured.
- Over 1000 developers are registered in the CATS-US Developer support program. Over 500 are

IN THIS ISSUE

2.0 Compatibility Problem Areas
Carolyn Scheppner, Bryce Nesbitt, Peter Cherna, and Darren Greenwald

Notification
Ewout Walraven

Amiga Font Scaling and Aspect Ratio
John Orr

Keeping Time--Interval Timers in Amiga UNIX
David Miller

registered in the support program in Europe. This is a significant increase over last year.
- Commodore continues to be profitable, and just closed the fiscal year at over $1 billion, for the first time since the '64 days.
- In the next month or two, we expect to ship the 3 millionth Amiga!

Release 2 Enhancer Kits

One of the most popular announcements at the Amiga Developers Conference was Commodore's plans to make AmigaDOS Release 2 Enhancer Kits available starting in October. In the United States, a full update to Release 2 (including Release 2 Kickstart ROMs) will be made available through service centers. Manufacturing plans include a controlled production phase-in of Release 2 on all current Amiga models.

Commodore has worked closely with Amiga software developers to ensure compatibility of software with Release 2. Over 300 Amiga software titles have signed up for Commodore's "Release 2 Compatible" sticker program.

CDTV Sessions

Members of Commodore's Special Projects Group spoke at eight DevCon sessions. Gail Wellington, Director of Special Projects, reviewed current marketing plans with the developers. Promotion in the U.S. will include a major launch in key test markets, the largest being San Francisco. An aggressive television and print advertising run is scheduled along with a direct mail campaign. A continuing geographically targeted marketing campaign is being planned. It is expected that 10-15 major markets will be launched in the U.S. by Christmas.

A similar campaign, on a national level, is being planned for the United Kingdom. During a recent U.K. promotional campaign fully 80% of respondents who visited the store(s) carrying CDTV purchased a unit.

Gail also announced that Commodore is bundling *Groliers New Electronic Encyclopedia* and Psygnosis' *Lemmings* with every CDTV unit in the U.S. *Groliers New Electronic Encyclopedia* is one of the best selling CD-based titles, according to industry analysts. *Lemmings* has been a runaway hit for Psygnosis on the Amiga platform.

Perhaps in conjunction with this announcement, Gail (and an entourage including Lauren Brown, Jay Denebeim, Matt Dillon, Harry Evangelou, Chris Green, Robin Lapasham, Dale Larson, Bryce Nesbitt and John Toebes) made an impromptu appearance as Lemmings at the Denver DevCon's Developers' Showcase. The group, appropriately decked out in green wigs and blue jumpers, staged the performance accompanied by a tape of the *Lemmings* theme. They blocked, climbed, floated and eventually were nuked to the delight of all present.

You Ask Us: Open Forum

One of the perennially popular and well attended DevCon sessions is the open forum that officially closes the conference. This year was no exception. This session provides an arena for developers to pose questions to a panel of Commodore representatives from CATS, Engineering, Special Projects and Marketing. Forty-eight percent of attendees rated the session Excellent; 43% rated it Very Good.

At this year's You Ask Us, Jeff Scherb, in a David Lettermanesque intro, listed the "Top Ten Reasons Why I'm Not 2.0 Compatible." They are:

10. 37.68 is my lucky number.
9. Oops, I thought you meant OS/2!
8. Trust Me! My code works.
7. The code is in the mail.
6. I *like* the Guru.
5. I can't live without Notepad.
4. It only got two stars in *.info*.*
3. Larry "Bud" Melman is still on 1.2.
2. Bryce doesn't call me any more.
1. The dog ate my RKMs.

*In an effort to ward off hate mail, the editors of *.info* have asked us to point out that this is a joke -- they don't actually *review* operating systems, and they never gave Release 2 only two stars.

Q: Is it possible to have one IDCMP *port for two windows on two screens?*

A: Yes. The standard technique for sharing one IDCMP port is:

```
sharedport = CreatePort( ... );
    ...
newwin.IDCMPFlags = 0;
    ...
win = OpenWindow( &newwin );
win->UserPort = sharedport;
ModifyIDCMP( win, DESIRED_IDCMPFLAGS );
    ...
/* Open other windows the same way */
    ...
/* You must use CloseWindowSafely() to close
windows with shared IDCMP ports -- CloseWindow()
alone is unsafe... */
CloseWindowSafely( win );
```

Notice that this code uses the function CloseWindowSafely() (see the CloseWindow() *Autodoc* for the code). This function is specially designed to clean up after a window that shares an IDCMP port either with a window or some other inter-process communications customer. *Don't modify the loop in the CloseWindow() Autodoc! It is correct.* Some people think they see a bug there (one iteration too many), but they're wrong.

Q: How do I provide a Zorro II 24-bit DMA device with a few bytes of memory on any machine?

A: Call the AllocMem() function using the new (for 2.0) MEMF_24BITDMA flag defined in <exec/memory.h>. (It is much like MEMF_CHIP and MEMF_FAST only for 24BITDMA...)

Q: but what if I'm using a 1.3 system?

A: If you call AllocMem() with the MEMF_24BITDMA flag, the pre-2.0 AllocMem() will still check for that flag. You could write a utility that, under pre-2.0 systems, adds the MEMF_24BITDMA flag to a memory header's MH_ATTRIBUTES field of the 24-bit memory areas.

Q: With some programs I get a lot of enforcer *hits (write-long) at addresses $180, $184, $188, etc. with PC in ROM at location $F830C. Why?*

A: See the *Autodoc* for *exec.library/Alert*. This is where exec stores the processor registers when there is an alert caused by a processor trap. You should be able to make some guesses about what's wrong with your program by looking at the register values being written. The Guru Number is in D7.

Q: Commodore has publicly stated that applications should not *make specific use of the PMMU. Is this correct?*

A: Yes. While tools/hacks sometimes use the PMMU (such as *Enforcer*) they can not be expected to work when the system itself starts to use the PMMU. The PMMU is a supervisor-space feature of the CPU and, for it to do what it does, it must remain a system controlled resource. If you wish to do MMU-related things in an OS and processor compatible manner, use the function calls available in 2.04 (such as CachePreDMA(), CacheControlU(), etc). These will do their best to "do the right thing" on all OS/CPU combinations.

More complex usages (private MMU tables, etc) are the domain of the OS and of various hacks (but often highly useful hacks) that may not work with future OS/CPU combinations.

Q: When opening a BORDERLESS *window under 2.0, do you have to also make it* BACKDROP *to stop it from having Borders?*

A: A window has borders unless *all* of the following are true:

1. It is declared BORDERLESS.
2. It has no title (NewWindow.Title = NULL, not "").
3. It has no user gadgets in the border (i.e. gadgets with xxxBORDER set).
4. It has no system gadgets in the border (i.e. no close gadget, no size gadget, no depth gadgets, etc.).

Now the difference with 2.0 is that the NewLook borders are rendered when a window goes active or inactive. Under 1.3, the borders are not re-rendered when the window becomes inactivate.

Some programs have a close gadget, and when they open their window, they write over their border. They may also have to overwrite it when they get an ACTIVEWINDOW IDCMP message. Under 2.0, this trick/kludge isn't enough.

BACKDROP does not enter into the picture.

The real trouble is that Intuition does not directly support putting gadgets like the close gadget into borderless windows. It never did, and the kludge that makes it basically work under 1.3 is not sufficient under 2.0.

In general, It's not a good idea, stylistically, to have borderless windows that are not backdrop windows, since that may make it difficult for the user to see the window bounds against other windows behind it.

Q: Some of the format specifiers for the dos.library function VFWriteF() contain a numerical value for field lengths. The documentation states that the numerical value is in base 36! Why is base 36 used?

A: It is a remnant from BCPL. It is the base you get when using all of the alphanumeric characters.

Q: I am trying to get the number of display rows from the Workbench screen. What is the proper procedure for obtaining the number of Rows?

A: Under 2.0, you should use the Graphics Display Database to get this information.

```
struct DimensionInfo MyDimInfo;
struct Screen *WBenchScreen;
WBenchScreen = LockPubScreen("Workbench");
id=GetVPModeID(&(WBenchScreen->ViewPort));
GetDisplayInfoData(NULL,&MyDimInfo,
        sizeof(MyDimInfo), TAG_DIMS, id);
```

MyDimInfo will now contain the sizes and positions for Nominal, Video Overscan, Text Overscan, and Standard overscan. Text and Standard are from the user's preferences settings.

Q:How do I rename a disk?

A: Under 1.3, you have to send the Action_Rename_Disk packet to the disk's filesystem. See the *AmigaDOS Manual* for more details on this packet. To find out where to send the packet, use DeviceProc()on the old name (plus the ":") or the device name (i.e.: "df0:").

Under 2.0, you can use the new Relabel() function of *dos.library*. That function basically sends an Action_Rename_Disk packet.

2.0 Compatibility Problem Areas

by Carolyn Scheppner, Bryce Nesbitt, Peter Cherna, and Darren Greenwald

Release 2.04 is finished and soon will be available in Enhancer Packs. If your products are not yet compatible, *your time is up*. If you are developing new software, you need to avoid the compatibility traps. This comprehensive list of 2.0 compatibility problem areas can help you avoid compatibility problems. Note that this document is not identical to the one published in the 1991 DevCon notes. There are several extra problems covered here that weren't mentioned in the 1991 DevCon notes.

General Compatibility Problem Areas

One sure fire way to write incompatible software is to fail to follow the Amiga programming guidelines listed in the beginning of your *Amiga ROM Kernel and Amiga Hardware* manuals. Please read the guidelines and follow them!

The following improper Amiga programming practices are likely to fail on new ROMs or hardware:

- ❑ Calling ROM code directly.
- ❑ Directly or indirectly reading or writing random memory addresses or low memory (especially location zero) due to improperly initialized pointers or structures. Use *Mungwall* and *Enforcer* when writing and testing your code!
- ❑ Assuming addresses/location/amounts of RAM or system structures.
- ❑ Requiring all free RAM.
- ❑ Mishandling 32-bit addresses. For example, using signed math or signed comparisons on addresses, or doing a BOOL or WORD test to determine if a pointer is non-zero.
- ❑ Overwriting memory allocations. With 32-bit addresses, a 1-byte overwrite of a string array can wipe out the high byte of a pointer or stack return address. This bug could go unnoticed on a 24-bit address machine (eg. A500,A2500, etc) but crash the system or cause other problems on an A3000.
- ❑ Shaving stack size too close. System function stack usage changes with each OS release.

- ❑ Improper flags or garbage in system structures. A bit that means nothing under one OS may drastically change the behavior of a function in a newer version of the OS. Clear structures before using, and use correct flags.

- ❑ Passing garbage in previously unused upper bytes of function arguments (for example - the upper word of the ULONG AvailFonts() Flags parameter).

- ❑ Improper register or condition code handling. Do not assume registers D0-D1/A0-A1 are preserved after system calls! Some function calls happen to preserve some registers. This can change in any revisions of the OS. In some cases we have modifed the values returned in registers to keep certain applications from failing under 2.0. We do not guarantee those modifications will remain in place. Condition codes are also in an undefined state on the return from a system call. Assembler code must test (TST,MOVE,CMP,etc.) D0 results before branching on condition codes. Use *Scratch* by Bill Hawes (via the *scratchall* script) to catch scratch register misuse in assembler code.

- ❑ Misuse of function return values. Use function prototypes and read the Autodocs for the functions you are using. Some system functions return just success or failure, or nothing at all (void). In such cases, the value which the function happens to return must not be used except as it is documented.

- ❑ Calling system library functions from assembler without placing the library base pointer in A6. All system functions may assume that their library's base pointer is in A6. A function's need to reference its library base can change in different OS revisions.

- ❑ Depending on unsupported side effects or undocumented behavior. Be sure to read the RKM chapters, Autodocs, and include file comments.

- ❑ Poking/peeking system private structures. Do not poke or peek any system structure unless told to do so in official Commodore documentation.

- ❑ Assuming current choices, configurations or initial values. If the current possibilities are A, B, or C, do not assume C if it isn't A or B. Check specifically for the choices currently implemented, and provide default behavior for unexpected values.

- ❑ Failure to properly allocate resources before using them.

- ❑ Failure to properly close/deallocate resources.

- ❑ Improper reading/writing of hardware registers. You must mask out bits you are not interested in on reads, and write undefined bits as zero.

- ❑ Assuming initial values of hardware registers. If you are going direct to the hardware, do not depend on the initial values of any hardware registers. The settings may not be the same on different versions of the OS or from boot to boot. Always set up all of the hardware registers that affect your code.

❑ Processor speed dependencies such as software delay loops.

❑ Processor instruction dependencies. Do not use instructions which are privileged on any Motorola 68xxx family processor. Do not use CLR on a hardware register which is triggered by any access (use MOVE #0 instead). The 68000 CLR instruction performs two accesses (Read, then Write). The 68020 and higher CLR instruction performs just one access.

❑ Depending on or failing to account for cache or prefetch effects. Self-modifying or trackdisk-loaded code requires cache flushes (see the exec.library/CacheClearU() function).

Amiga debugging tools such as *Enforcer*, *Mungwall*, and *Scratch* can find many program bugs that may affect compatibility. A program that is *Enforcer/Mungwall/Scratch* clean stands a much better chance of working well under current and future versions of the OS. These tools are on the Denver/Milano DevCon disks. *Enforcer* and *Mungwall* are also on the kickfile disks.

2.0 Changes That Can Affect Compatibility

There are several 2.0-specific areas where OS changes and enhancements can cause compatibility problems for some software.

Exec

❑ *Do not* jump to location $FC0002 as part of performing a system RESET. Many RESET functions jumped to what was the start of the ROM under 1.3. The 2.0 ROM is twice the size. We've added a *temporary* compatibility hack called "Kickety-Split" to the 2.04 Kickstart ROM. The ROM is split into two halves with a redirecting jump at $FC0002. This hack does *not* appear on the A3000 and due to space considerations will *not* appear on future machines.

❑ Everything has moved.

❑ The Supervisor stack is not in the same place as it was under 1.3. This has caused problems for some games that completely take over the Amiga. If your program goes into Supervisor mode, you must either respect allocated memory or provide your own Supervisor stack when taking over the machine.

❑ ExecBase is moved to expansion memory if possible. Previously, ExecBase would only end up in one of two fixed locations. As a result, ColdCapture may be called after expansion memory has been configured. Great pains were taken to make this compatible.

❑ Exception/Interrupt vectors may move. This means the 68010 and above Vector Base Register (VBR) may contain a non-zero value. Poking assumed low memory vector addresses may have no effect. You must read the VBR on 68010 and above to find the base.

- No longer tolerant of wild Forbid() counts. Under 1.3, sometimes this bug could go unnoticed. Make sure that all Forbid's are matched with one and only one Permit (and vice versa).

- When an Exec device gets an IORequest, it must validate io_Command. If the io_Command is 0 or out of range, the device must return IOERR_NOCMD and take no other action. The filesystem now sends new commands and expects older devices to properly ignore them.

- A 2.0 fix to task-switching allows a busy task to properly regain the processor after an interrupt until either its quantum (4 vertical blanks) is up or a higher priority task preempts it. This can dramatically change the behavior of multitask programs where one task busyloops while another same-priority task Wait()s. See *Task Switching* in the *Additional Information* section at the end of this document.

Expansion

- ExpansionBase is private - use FindConfigDev().

- Memory from contiguous cards of the same memory type is automatically merged into one memory pool.

Strap

- *romboot.library* is gone.

- *audio.device* cannot be OpenDevice()ed by a boot block program because it is not yet InitResident()ed. If OpenDevice() of *audio.device* fails during strap, you must FindResident()/InitResident() *audio.device*, and then try OpenDevice() again.

- Boot from other floppies (+5,-10,-20,-30) is possible.

- Undocumented system stack and register usage at Diag and Boot time have changed.

DOS

- DOS is now written in C and assembler, not BCPL. The BCPL compiler artifact which caused D0 function results to also be in D1 is gone. 2.0 compatibility patches which return some DOS function results in both D0 and D1 are not guaranteed to remain in the next release. Fix your programs! Use *Scratch* to find these problems in your code.

- DOS now has a real library base with normal LVO vectors.

- Stack usage has all changed (variables, direction).

- New packet and lock types. Make sure you are not passing stack garbage for the second argument to Lock().

- Process structure is bigger. "Rolling your own" Process structure from a Task fails.

- Unless documented otherwise, you must be a process to call DOS functions. DOS function dependence on special process structures can change with OS revisions.

Audio.device

- Now not initialized until used. This means low memory open failure is possible. Check your return values from OpenDevice(). This also means *audio.device* cannot be opened during 2.0 Strap unless InitResident()ed first. If OpenDevice() of *audio.device* fails during strap, you must FindResident()/InitResident() *audio.device*, and then try OpenDevice() again. There will be a small memory loss (until reboot) generated by the first opener of *audio.device* or *narrator.device* (memory used in building of *audio.device*'s base).

Gameport.device

- Initial state of hardware lines may differ.

Serial.device

- Clears io_Device on CloseDevice() (since 1.3.2)

Timer.device

- The most common mistake programmers make with *timer.device* is to send off a particular timerequest before the previous use of that timerequest has completed. Use *IO_Torture* to catch this problem.

- IO_QUICK requests may be deferred and be replied as documented.

- VBLANK timer requests, as documented, now wait at least as long as the full number of VBlanks you asked for. Previously, a partial vertical blank could count towards your requested number. The new behavior is more correct and matches the docs, but it can cause VBlank requests to now take up to 1 VBlank longer under 2.0 as compared to 1.3. For example, a 1/10 second request, may take 6-7 Vblanks instead of 5-6 VBlanks, or about 15% longer.

Trackdisk

- ❑ Private trackdisk structures have changed. See *trackdisk.doc* for a compatible REMCHANGEINT.
- ❑ Buffer is freeable, so low memory open failure is possible.
- ❑ *Do not* disable interrupts (any of them) then expect trackdisk to function while they are disabled.

CIA Timers

- ❑ System use of CIA timers has changed. Don't peek timers you think the system is using in a particular manner.
- ❑ Don't depend on initial values of CIA registers.
- ❑ Don't mess with CIABase. Use *cia.resource.*
- ❑ If your code requires hardware level CIA timers, allocate the timers using *cia.resource* AddICRVector()! Very important! Operating system usage of the CIA timers has changed. The new 2.0 *timer.device* ("Jumpy the Magic Timer Device") will try to jump to different CIA's so programs that properly allocate timers will have a better chance of getting what they want. If possible, be flexible and design your code to work with whatever timer you can successfully allocate.
- ❑ OS usage of `INT6` is increasing. Do not totally take over `INT6`, and do not terminate the server chain if an interrupt is not for you.

Other Hardware Issues

- ❑ Battery-backed clock is different on A3000. Use *battclock.resource* to access the real-time clock if *battclock.resource* can be opened.
- ❑ A 68030 hardware characteristic causes longword-aligned longword writes to allocate a valid entry in the data cache, ***even if*** the hardware area shouldn't be cached. This can cause problems for IO registers and shared memory devices. To solve this: 1) Don't do that 2) Flush the cache or 3) Use *Enforcer Quiet.* See the Motorola 68030 manual under the description of the Write Allocate bit (which must be set for the Amiga to run with the Data Cache).

Intuition

- Private IBase variables have moved/changed. Reading them is illegal. Writing them is both illegal and dangerous.

- Poking IBase MaxMouse variables is now a no-op, but please stop poking when Intuition version is >35.

- If you are opening on the Workbench screen, be prepared to handle larger screens, new modes, new fonts, and overscan. Also see Font compatibility information.
- Screen TopEdge and LeftEdge may be negative.

- LeftAmiga-Select is used for dragging large screens. Do not use LeftAmiga-key combinations for application command keys. The LeftAmiga key is reserved for system use.

- For compatibility reasons, GetScreenData() lies if the Workbench screen is a mode only available after release 1.3. It will try to return the most sensible mode that old OpenScreen() can open. This was necessary to keep applications that cloned the Workbench screen from having problems. To properly handle new modes, see LockPubScreen() and GetVPModeID(), and the SA_DisplayID tag for OpenScreenTags().

- Using combined `RAWKEY` and `VANILLAKEY` now gives `VANILLAKEY` messages for regular keys, and `RAWKEY` messages for special keys (fkeys, help, etc.)

- Moving a `SIMPLE_REFRESH` window does not necessarily cause a `REFRESHWINDOW` event because layers now preserves all the bits it can.

- Sizing a `SIMPLE_REFRESH` window will not clear it.

- `MENUVERIFY/REQVERIFY/SIZEVERIFY` can time out if you take too long to ReplyMsg().

- Menu-key equivalents are ignored while string gadgets are active.

- You can't type control characters into string gadgets by default. Use CTRL-Amiga-char to type them in or use IControl Prefs to change the default behavior.

- Width and Height parameters of AutoRequest are ignored.

- New default colors, new gadget images.

- JAM1 rendering/text in border may be invisible gadgets over default colors.

- The cursor for string gadgets can no longer reside outside the cleared container area. If your gadget is (for example) 32 pixels wide, with MaxChars of 4, then all 32 pixels will be cleared, instead of just 24, as was true in 1.3.

- Applications and requesters that fail to specify desired fonts will get user 2.0 Font Pref fonts that may be much larger or proportional in some cases. Screen and window titlebars (and their gadgets) will be taller when accomodating a larger font. Applications which open on the Workbench screen must adapt to variable size titlebars. Any application which accepts system defaults for its screen, window, menu, Text or IntuiText fonts must adapt to different fonts and titlebar sizes. String gadgets whose height is too small for a font will revert to a smaller ROM font. There are now 2 different user-specifiable default system fonts which affect different Intuition features. This can lead to mismatches in mixed gadgets and text. For more information on where various system fonts come from and how they can be controlled, see *Intuition Fonts* in the *Additional Information* section at the end of this document.

- Don't modify gadgets directly without first removing them from the gadget list, unless you are using a system function designed for that purpose, such as NewModifyProp() or SetGadgetAttrs().

- Don't rely on NewModifyProp() to fully refresh your prop gadget after you've changed values in the structure. NewModifyProp() will only correctly refresh changes which were passed to it as parameters. Use Remove/Add/RefreshGList() for other kinds of changes.

- Custom screens must be of type `CUSTOMSCREEN` or `PUBLICSCREEN`. Other types are illegal. One application opens their screen with NewScreen.Type = 0 (instead of `CUSTOMSCREEN`, 0x0F). Then, when they open their windows, they specify NewWindow.Type of 0 and NewWindow.Screen of `NULL`, instead of Type=`CUSTOMSCREEN` and Screen=(their screen). That happened to work before, but they are now broken.

- Referencing IntuiMessage->IAddress as a Gadget pointer on non-Gadget IDCMP messages, or as a Window pointer (rather than looking at the proper field IntuiMessage->IDCMPWindow) may now cause *Enforcer* hits or crashes. The IAddress field used to always contain a pointer of some type even for IDCMP events for which no IAddress value is documented. Now, for some IDCMP events, IAddress may contain other data (a non-address, possibly an odd value which would crash a 68000 based system).

- Using Intuition flags in the wrong structure fields (for example, using `ACTIVEWINDOW` instead of `ACTIVATE`). To alleviate this problem, 2.0 has introduced modern synonyms that are less confusing than the old ones. For example, `IDCMP_ACTIVEWINDOW` and `WFLG_ACTIVATE`. This particular example of confusion (there are several) was the nastiest, since `IDCMP_ACTIVEWINDOW`, when stuffed into NewWindow.Flags, corresponds numerically to `WFLG_NW_EXTENDED`, which informs Intuition that the NewWindow structure is immediately followed by a TagList, which isn't there! Intuition does some validation on the tag-list pointer, in order to partially compensate. To make your compiler use the new synonyms only, add this line to your code before Intuition include files: `#define INTUI_V36_NAMES_ONLY`.

- Do not place spaces into the StringInfo->Buffer of a LONGINT string gadget. Under 1.3, it turned out that worked, but under 2.0, the validation routine that checks for illegal keystrokes looks at the contents for illegal (i.e. non-numeric) characters, and if any are found assumes that the user typed an illegal keystroke. The user's only options may be shift-delete or Amiga-X. Use the correct justification instead.

- If you specify NULL for a font in an IntuiText, don't assume you'll get Topaz 8. Either explicitly supply the font you you need or be prepared to size accordingly. Otherwise, your rendering will be wrong, and the user will have to reset his Preferences just to make your software work right.

- Window borders are now drawn in the screen's DetailPen and BlockPen rather than the Window's pens. For best appearance, you should pass a SA_Pens array to OpenScreen(). This can be done in a backwards compatible manner with the ExtNewScreen structure and the NS_EXTENDED flag.

- The system now renders into the full width of window borders, although the widths themselves are unchanged.

- Window borders are filled upon activation *and* inactivation.

- Window border rendering has changed significantly for 2.0. Note that the border dimensions are unchanged from 1.x (Look at window->BorderLeft/Top/Width/Height if you don't believe us!). If your gadget intersects the border area, although it may have looked OK under 1.3, a visual conflict may occur under 2.0. If Intuition notices a gadget which is substantially in the border but not declared as such, it treats it as though it were (this is called "bordersniffing"). Never rely on Intuition to sniff these out for you; always declare them explicitly (see the Gadget Activation flags GACT_RIGHTBORDER etc.). See *Intuition Gadgets and Window Borders* in the *Additional Information* section at the end of this article.

Preferences

- Some old struct Preferences fields are now ignored by SetPrefs (for example FontHeight). SetPrefs also stops listening to the pointer fields as soon as a new-style pointer is passed to Intuition (new-style pointers can be taller or deeper).

- Preferences ViewX/YOffset only applies to the default mode. You cannot use these fields to move the position of all modes.

- The Preferences LACEWB bit is not necessarily correct when Workbench is in a new display mode (akin to GetScreenData()).

Workbench

- The Workbench GUI now has new screen sizes, screen top/left offsets, depths, modes, and fonts.

- Default Tool now searches paths.

- New Look (boxed) icons take more space.

- Do not use icons which have more 1bits set in PlanePick than planes in the ImageData (one IFF-to-Icon utility does this). Such icons will appear trashed on deeper Workbenches.
- New Look colors have black and white swapped (as compared to 1.3).

- The Workbench screen may not be open at startup-sequence time until some output occurs to the initial Shell window. This can break startup-sequence-started games that think they can steal WB's screen bitplanes. Do not steal the WB screen's planes (For compatibility, booting off pre-2.0 disks forces the initial screen open. This is not guaranteed to remain in the system). Use startup code that can detach when RUN (such as *cback.o*) and use CloseWorkbench() to regain the screen's memory.

  Under 1.3 the Workbench Screen and initial CLI opened before the first line in *s:startup-sequence*. Some naughty programmers, in an attempt to recover memory, would search for the bitplane pointers and appropriate the memory for their own use. This behavior is highly unsafe.

  By default 2.0 opens the initial CLI on the first _output_ from the *s:startup-sequence*. This allows screen modes and other parameters to be set before the user sees the screen. However, this broke so many programs that we put in the "silent-startup" hack. A disk installed with 1.3 install opens the screen as before. A disk installed under 2.0 opens silently.
  Never steal the Workbench bitplanes. You don't know where they are, how big they are, what format they may be in, or even if they are allocated. Recovering the memory is a bit tricky.
  Under 2.0: Simply avoid *any* output from your *s:startup-sequence*. If your program opens a screen it will be the first screen the user ever sees. Note that if ENDCLI is ever hit, the screen will pop open.

  Under 1.3: After ENDCLI, call the CloseWorkbench() function to close the screen. This also works under 2.0. Loop on CloseWorkbench() with a delay between loops. Continue looping until CloseWorkbench() succeeds or too much time has passed. Note that a new program called *EndRun* is available for starting non-returning programs from the *startup-sequence*. *EndRun* will reduce memory fragmentation and will close Workbench if it is open. *EndRun.lzh* will be available in Commodore's Amiga listings area on BIX.

Layers

- Use NewLayerInfo() to create, not FattenLayerInfo(), ThinLayerInfo(), InitLayers().

- Simple-refresh preserves all of the pixels it can. Sizing a `SIMPLE_REFRESH` window no longer clears the whole window.

- Speed of layer operations is different. Don't depend on layer operations to finish before or after other asynchronous actions.

Graphics

- Do not rely on the order of Copper list instructions. For example, 2.0's MrgCop() builds different Copper lists to that of 1.3, by including new registers in the list (e.g. MOVE xxxx,DIWHIGH). This changes the positions of the other instructions. We know of one game that 'assumes' the BPLxPTRs would be at a certain offset in the Copper list, and that is now broken on machines running 2.0 with the new Denise chip.

- Graphics and layers functions which use the blitter generally return after STARTING the final blit. If you are mixing graphics rendering calls and processor access of the same memory, you must WaitBlit() before touching (or deallocating) the source or destination memory with the processor. For example, the Text() function was sped up for 2.0, causing some programs to trash partial lines of text.

- ColorMap structure is bigger. Programs must use GetColorMap() to create one.

- Blitter rtns decide ascend/descend on 1st plane only.

- Changing the display mode of an existing screen or viewport while it is open is still not a supported operation.

- GfxBase DisplayFlags and row/cols may not match Workbench screen.

- *Do not* hardcode modulo values - use BitMap->BytesPerRow.

- If the graphics *Autodocs* say that you need a TmpRas of a certain size for some functions, then you ***must*** make that the minimum size. In some cases, before 2.0, you may have gotten away with using a smaller TmpRas with some functions (for example Flood()). To be more robust, Graphics now checks the TmpRas size and will fail the function call if the TmpRas is too small.

- ECS chips under 2.0 use a different method of generating displays. The display window registers now control DMA.

- LoadRGB4() used to poke colors into the active copperlist with no protection against deallocation of that copperlist while it was being poked. Under 2.0, semaphore protection of the copperlist was added to LoadRGB4(). This semaphore protection makes it totally incorrect and extremely dangerous to call LoadRGB4() during an interrupt. The general symptom of this problem is that a system deadlock can be caused by dragging one screen up and down while another is cycling. Color cycling should be performed from within a task, not an interrupt. Note that in general, the only functions which may be safely called from within an interrupt are the small list of Exec functions documented in the "Exec: Interrupts" chapter of *ROM Kernel Manual: Libraries and Devices.*

Fonts

- Some font format changes (old format supported).

- Private format of .font files has changed (use FixFonts to create).

- Default fonts may be larger, proportional.

- Topaz is now sans-serif.

- Any size font will be created via scaling as long as TextAttr.Flags `FPF_DESIGNED` bit is *not* set. If you were asking for an extreme size, like size 1 to get smallest available, or 999 to get largest available, you will get a big (or very very small) surprise now.

- *Do not use* -1 for TextAttr.Flags or Styles, nor as the flags for AvailFonts (one high bit now causes AvailFonts to return different structures). Only set what you know you want. A kludge has been added to the OS to protect applications which currently pass -1 for AvailFonts flags.

CLI / Shell

- Many more commands are now built-in (no longer in C:). This can break installation scripts that copy C:commandname, and programs that try to Lock() or Open() C:commandname to check for the command's existence.

- The limit of 20 CLI processes is gone and the DOSBase CLI table has changed to accomodate this. Under V36 and higher, you should use new 2.0 functions rather than accessing the CLI table directly.

- Shell windows now have Close Gadgets. The EOF character is passed for the Close Gadget of a Shell. This is -1L with CON: getchar(), and the Close Gadget raw event ESC seq with RAW:.

- Shells now use the simple-refresh character-mapped console (see Console notes).

Console

- By default, CON: now opens SIMPLE_REFRESH windows using the V36/V37 console character mapped mode. Because of some differences between character mapped consoles, and SMART_REFRESH non-mapped consoles, this may cause incompatibilities with some applications. For example, the Amiga private sequences to set left/top offset, and set line/page length behave differently in character mapped console windows. The only known work-around is to recompile asking for a CON: (or RAW:) window using the SMART flag.

- Simple refresh/character mapped console windows now support the ability to highlight, and *copy* text with the mouse. This feature, as well as *pasting* text should be transparent to programs which use CON: for console input, and output. *Pasted* text will appear in your input stream as if the user had typed it.

- While CONCLIP (see *s:startup-sequence*) is running, programs may receive "<CSI>0 v" in their input stream indicating the user wants to paste text from the clipboard. This shouldn't cause any problems for programs which parse correctly (however we know that it does; the most common problems are outputting the sequence, or confusing it with another sequence like that for FKEY 1 which is "<CSI>0~").

- The *console.device* now renders a ghosted cursor in inactive console windows (both SMART_REFRESH, and SIMPLE_REFRESH with character maps). Therefore, rendering over the console's cursor with *graphics.library* calls can *trash* the cursor; if you must do this, first turn off the cursor.

- Some degree of unofficial support has been put in for programs which use SMART_REFRESH console windows, and use *graphics.library* calls mixed with *console.device* sequences to scroll, draw text, clear, etc. This is *not* supported in SIMPLE_REFRESH windows with character maps, and is strongly discouraged in all cases.

- Closing an Intuition window *before* closing the attached *console.device* worked in the past; it will now crash or hang the machine.

- Under 1.2-1.3, vacated portions of a console window (e.g., areas vacated because of a clear, or a scroll) were filled in with the character cell color. As of V36 this is no longer true; vacated areas are filled in with the *global* background color which can be set using the SGR sequence "<ESC>[>##m" where ## is a value between 0-7. In order to set the background color under V36/V37, send the SGR to set background color, and a FormFeed to clear the screen.

- Note that SIMPLE_REFRESH character mapped consoles are immediately redrawn with the global background color when changed--this is not possible with SMART_REFRESH windows.

Additional Information

Task Switching

The 1.3 Kickstart contained two task switching bugs. After an interrupt, a task could lose the CPU to another equal priority task, even if the first task's time was not up. The second bug allowed a task who's time was up to hold on to the CPU either forever, or until a higher priority task was scheduled. Two busy-waiting tasks at high priority would never share the CPU. Because the *input.device* runs at priority 20, usually the effect of these bugs was masked out for low priority tasks. Because of the bugs, the ExecBase->Quantum field had little effect.

For 2.0, a task runs until either its Quantum is up, or a higher priority task preempts it. When the Quantum time is up, the task will now lose the CPU. The Quantum was set to 16/60 second for 1.3, and 4/60 second for 2.0.

In general, the 2.0 change makes the system more efficient by eliminating unnecessary task switches on interrupt-busy systems (for example, during serial input). However, the change has caused problems for some programs that use two tasks of equal priority, one busy-waiting and one Wait()ing on events such as serial input. Previously, each incoming serial character interrupt would cause task context switch allowing the event-handling task to run immediately. Under 2.0 the two tasks share the processor fairly.

Intuition Gadgets and Window Borders

If 2.0 Intuition finds a gadget whose hit area (gadget Left/Top/ Width/Height) is substantially inside the border, it will be treated as though it was declared in the border. This is called "bordersniffing". Gadgets declared as being in the border or detected by Intuition as being in the border are refreshed each time after the border is refreshed, and thus aren't clobbered.

Noteworthy special cases:

1) A gadget that has several pixels not in the border *is not* bordersniffed. An example would be an 18-pixel high gadget in the bottom border of a `SIZEBBOTTOM` window. About half the gadget will be clobbered by the border rendering.

2) A gadget that is not substantially in the border but has imagery that extends into the border cannot be sniffed out by Intuition.

3) A gadget that is substantially in the border but has imagery that extends into the main part of the window will be sniffed out as a border gadget, and this could change the refreshing results. A common trick to put imagery in a window is to put a 1x1 or 0x0 dummy gadget at window location (0,0) and attach the window imagery to it. To support this, Intuition will never bordersniff gadgets of size 1 x1 or smaller.

All these cases can be fixed by setting the appropriate `GACT_xxxBORDER` gadget Activation flag.

4) In rare cases, buttons rendered with Border structures and `JAM1` text may appear invisible under 2.0. We apologize, but there is nothing that can be done on our end, even if the application technically did nothing wrong.

Intuition Fonts

The following table shows where the Intuition gets its fonts from:

What you tell OpenScreen	Screen's Font	Windows' RPort's Font
A. NewScreen.Font = myfont	myfont	myfont
B. NewScreen.Font = NULL	GfxBase->DefaultFont	GfxBase->DefaultFont
C. {SA_Font, myfont}	myfont	myfont
D. {SA_SysFont, 0}	GfxBase->DefaultFont	GfxBase->DefaultFont
E. {SA_SysFont, 1}	Font Prefs Screen text	GfxBase->DefaultFont

Notes:

A and B are the options that existed in releases prior to V36.

C and D are new V36 tags that are equivalent to A and B respectively.

E is a NEW option for V36. The Workbench screen uses this option.

GfxBase->DefaultFont will always be monospace. This is the "System Default Text" from Font Preferences.

The "Screen Text" choice from Font Preferences can be monospace or proportional.

'myfont' can be any font of the programmer's choosing, including a proportional one. This is true under all releases of the OS.

The menu bar, window titles, menu-items, and the contents of a string gadget use the screen's font. The font used for menu items can be overridden in the item's IntuiText structure. Under V36 and higher, the font used in a string gadget can be overridden through the StringExtend structure. The font of the menu bar and window titles cannot be overridden. Because the 2.0 Workbench screen uses option E to specify its Screen font from the user's Screen font Preferences, applications which open windows on the Workbench screen may get very large or proportional fonts in their menu bars, window titles, menu-items and string gadgets.

To predict your window's titlebar height before you call OpenWindow():

```
topborder = screen->WBorTop + screen->Font->ta_YSize + 1
```

The screen's font may not legally be changed after a screen is opened.

Be sure the screen cannot go away on you. This is true if:

1) You opened the screen yourself.
2) You currently have a window open on the screen.
3) You currently hold a lock on this screen (see LockPubScreen()).

IntuiText rendered into a window (either through PrintIText() or as a gadget's GadgetText) defaults to the Window RastPort font, but can be overridden using its ITextFont field. Text rendered with the Text() *graphics.library* call appears in the Window RastPort font.

The Window's RPort's font shown above is the *initial* font that Intuition sets for you in your window's RastPort. It is legal to change that subsequently with SetFont().

Notification

by Ewout Walraven

File Notification is a form of interprocess communication available under Release 2.0. An application can ask a file system (like the RAM disk handler RAM:, df0:, df1:...) that supports notification to inform it whenever changes are made to a specific file or directory, making it easy for the application to react to such changes. The V37 ROM file system and the V37 and V36 RAM disk handler support file notification.

Under Release 2.0, the preferences control program, *IPrefs*, sets up notification on most of the preferences files in *ENV:sys*. If the user alters any of these files (which he/she normally does with a preferences editor), the system will notify *IPrefs* about the change. *IPrefs* will react to this notification by attempting to alter the user's environment to reflect the preference change. For example, if the user opens the *ScreenMode* preferences editor and alters the Workbench environment so that the Workbench screen should be a Hires NTSC screen, ScreenMode writes a file called *Screenmode.prefs* to the *ENV:sys* directory which happens to be in RAM:. Because *IPrefs* has set up notification on this file, the RAM disk file system will notify *IPrefs* of the change, *IPrefs* will read in the *Screenmode.prefs* file and will try to reset the Workbench screen so it is in Hires NTSC mode.

Notification allows very different applications to share common data files without knowing anything about each other. This has many possible uses in the Amiga's single user, multitasking environment. One possible use for notification is in a desktop publishing (DTP) package. The user can open the DTP package to layout a group of ILBMs, some structured drawings, and word processed text. When the user loads each of these, the DTP package sets up notification on each of their corresponding files. If the user loads an appropriate editor and changes any of the files on which the DTP package has set up notification, the DTP package will receive notification of these changes and can automatically re-import these files into the current DTP document without the user having to intervene. Another possible use for notification might be in a *make* utility. A *make* program for a compiler could set up notification on a set of source code and object files. If any of those files change, the *make* program will recompile and link the program, without the programmer having to intervene.

Setting up file notification on a file is easy. The StartNotify() function from *dos.library* starts notification on a file or directory:

```
BOOL StartNotify( struct NotifyRequest *notify );
```

StartNotify() returns DOSTRUE if the call is successful, or it returns DOSFALSE (for example, when the file's file system does not support notification). This function takes a pointer to an initialized NotifyRequest structure as its only argument (as defined in *<dos/notify.h>*):

```
struct NotifyRequest {
    UBYTE *nr_Name;     /* File/directory name for which you want notification */
    UBYTE *nr_FullName; /* Used by DOS. Do not use */
    ULONG nr_UserData;  /* For applications use */
    ULONG nr_Flags;     /* Flags indicating Signal or Message notification */

    union {
        /* Used for Message notification */
        struct {
            struct MsgPort *nr_Port;    /* Message port to receive messages on */
        } nr_Msg;
        /* Used for Signal notification */
        struct {
            struct Task *nr_Task;       /* The task to signal */
            UBYTE nr_SignalNum;         /* The signal number to use. */
            UBYTE nr_pad[3];
        } nr_Signal;
    } nr_stuff;

    ULONG nr_Reserved[4];           /* leave 0 for now */

    /* Used internally by handlers */
    ULONG nr_MsgCount;              /* number of outstanding messages */
    struct MsgPort *nr_Handler;     /* handler to send to (for EndNotify) */
};
```

This structure must not be altered by the application while notification is in effect!

The nr_Name field contains a pointer to the name of the file on which to set up notification. Currently, nr_Name has to be a file name and path containing a logical device name (for example df0:, work:, fonts:). The nr_FullName field is for the private use of the file system. Any other use of it is strictly prohibited. The nr_UserData field is available for an applications private use.

The nr_Flags field tells the file system which type of notification to set up, *message* or *signal*. When the file system uses message notification, it notifies an application by sending an Exec message. An application asks a file handler to notify it via an Exec message by setting the NRF_SEND_MESSAGE flag in nr_Flags. When the file system uses signal notification, it sets an Exec signal to notify an application. An application receives notification via a signal by setting the NRF_SEND_SIGNAL flag.

The nr_Flags field has two other flags, NRF_WAIT_REPLY and NRF_NOTIFY_INITIAL. The NRF_WAIT_REPLY tells the file handler not to send notification messages about a specific file/directory to an application if the application has not replied to a previous notification message about that specific file. This flag only applies to message notification. The NRF_NOTIFY_INITIAL flag tells the file handler to notify the application if the file exists when it sets up notification on the file. The flags for the nr_Flags field are defined in *<dos/notify.h>*.

The layout of the rest of the NotifyRequest structure depends on the type of notification. If the application is using message notification, it must supply the handler with a message port to send the notification messages. The NotifyRequest.nr_stuff.nr_Msg.nr_Port field contain the pointer to the message port that will receive the message notifications. If the application is using a signal for notification, it must supply a pointer to the task to signal and the number (not bit!) of the signal. In this case, the NotifyRequest.nr_stuff.nr_Signal.nr_Task field should contain the appropriate task pointer and the NotifyRequest.nr_stuff.nr_Signal.nr_SignalNum field should contain the signal number.

When a file handler uses message notification, it will send a NotifyMessage:

```
struct NotifyMessage {
    struct Message nm_ExecMessage;
    ULONG  nm_Class;              /* Class, will be NOTIFY_CLASS */
    UWORD  nm_Code;               /* Code, will be NOTIFY_CODE */
    struct NotifyRequest *nm_NReq; /* Pointer to the NotifyRequest you supplied */
    ULONG  nm_DoNotTouch;         /* private */
    ULONG  nm_DoNotTouch2;        /* private */
};
```

Message notification is especially useful if you are monitoring more than one file. It quickly enables you to find out which file/directory caused this message by either comparing the NotifyRequest structure returned in nm_NReq with the one you sent in the StartNotify() function, or by reading the NotifyRequest's nr_UserData field. Because the NotifyMessage's nm_Class and nm_Code fields contain values that distinguish it from other types of messages, you can use an already allocated message port (from a window for example) to receive notification messages.

To end notification on a file, use the *dos.library* function EndNotify():

```
void EndNotify( struct NotifyRequest *notify );
```

An application must call this function for each of its successful StartNotify() calls. This function takes one parameter, a pointer to the NotifyRequest structure that the application used to initiate the notification. In the case of message notification, EndNotify() will remove all pending notify messages from your message port. After calling this function, it is safe for the application to change or free the NotifyRequest structure. The application may also remove the message port or free the signal bit.

A file handler should send notification when it receives any of the following packets (from *<dos/dosextens.h>*) about the notification file or directory:

```
ACTION_RENAME_OBJECT
ACTION_RENAME_DISK
ACTION_CREATE_DIR
ACTION_DELETE_OBJECT

ACTION_WRITE
ACTION_FINDUPDATE
ACTION_FINDOUTPUT
ACTION_SET_FILE_SIZE

ACTION_SET_DATE
```

The first four packets will cause notification immediately. The second four packets will cause notification when the notification file is closed. The last packet, ACTION_SET_DATE, should cause notification immediately, but due to a bug in the V37 ROM file system, only the RAM disk's file handler (RAM:) will send notification.

Notice that some of the packets that trigger a notification are sent by a process when it is trying to create a new file or directory. A file system that supports notification should be able to set up notification on a file or directory that does not currently exist. A file system should send notification when it creates that file or directory.

Note however that, although notification on directories is part of the OS in release 2.0, it does not work correctly. In the ROM file system, *directory notification only works if that directory exists when notification is set up.* If your application tries to set up notification on a ROM file system directory before the directory exists, *your application will never receive notification about that directory.* If notification is set up for a directory in RAM:, you will only be informed when that directory is created or deleted and when files are created in that directory, not when files are changed or deleted.

When implementing notification in your application, there are several things to remember. Not every file system supports notification, in particular, most network file systems will not support notification. For this reason, no application should require notification to function.

At the end of this article are two examples for file notification. *SignalNotification.c* implements signal notification on a single target. *MessageNotification.c* shows how to start message notification on multiple targets. Note that these examples are not linked with startup code (like *c.o*). Although these examples do set up SysBase and DosBase to gain access to *exec.library* and *dos.library*, the examples do not handle the startup message (WBenchMsg) that Workbench sends when it launches an application, *so do not run these examples from Workbench.*

```
;/* SignalNotification.c - Compiled with SAS/C 5.10a
lc -cfis -v -d0 -b0 -j73 SignalNotification.c
blink from SignalNotification.o to SignalNotification lib lib:amiga.lib
quit
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <dos/dosasl.h>
#include <dos/notify.h>
#include <dos/rdargs.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

/* Use pragmas if you've got them */

static UBYTE   *VersTag = "\0$VER: SignalNotification 37.1 (09.07.91)";
struct Library *DOSBase;
struct ExecBase *SysBase;

VOID            main(VOID);

VOID
main(VOID)
{

    struct RDArgs  *readargs;
    LONG            rargs[2];
    struct NotifyRequest *notifyrequest;
    UBYTE          *filename;
    ULONG           signr, signal;

    /* To appease amiga.lib */
    SysBase = (*((struct Library **) 4));

    /* Fail silently if < 37 */
    if (DOSBase = OpenLibrary("dos.library", 37))
    {
        /* See the DOS Autodocs for more information about ReadArgs() */
        if (readargs = ReadArgs("Filename/A", rargs, NULL))
        {
            filename = (UBYTE *) (rargs[0]);

            /* Allocate a NotifyRequest structure */
            if (notifyrequest = AllocMem(sizeof(struct NotifyRequest), MEMF_CLEAR))
            {
                /* And allocate a signalbit */
                if ((signr = AllocSignal(-1L)) != -1)
                {

                    /* Initialize notifcation request */
                    notifyrequest->nr_Name = filename;
                    notifyrequest->nr_Flags = NRF_SEND_SIGNAL | NRF_NOTIFY_INITIAL;
                    /* Signal this task */
                    notifyrequest->nr_stuff.nr_Signal.nr_Task =
                            (struct Task *) FindTask(NULL);
                    /* with this signal bit */
                    notifyrequest->nr_stuff.nr_Signal.nr_SignalNum = signr;

                    if ((StartNotify(notifyrequest)) == DOSTRUE)
                    {
                        /* Loop until Ctrl-C to exit */
                        for (;;)
                        {
                            signal = Wait(1L << signr | SIGBREAKF_CTRL_C);
                            if (signal & (1L << signr))
                                VFPrintf(Output(), "Notification signal!\n", NULL);
                            if (signal & SIGBREAKF_CTRL_C)
                            {
                                EndNotify(notifyrequest);
                                PrintFault(ERROR_BREAK, NULL);
                                break;
                            }
                        }
                    }
                    else
                        PrintFault(ERROR_NOT_IMPLEMENTED, NULL);    /* most logical */

                    FreeSignal(signr);
                }
                else
                    VFPrintf(Output(), "No signal available\n", NULL);
                FreeMem(notifyrequest, sizeof(struct NotifyRequest));
            }
            else
                PrintFault(ERROR_NO_FREE_STORE, NULL);

            FreeArgs(readargs);
        }
        else
            PrintFault(IoErr(), NULL);
        CloseLibrary(DOSBase);
    }
}




;/* MessageNotification.c - Compiled with SAS/C 5.10a:
lc -cfis -v -d0 -b0 -j73 MessageNotification.c
blink from MessageNotification.o to MessageNotification lib lib:amiga.lib
quit
*/

#include <exec/types.h>
#include <exec/memory.h>
#include <exec/lists.h>
#include <exec/nodes.h>
#include <dos/dos.h>
#include <dos/dosasl.h>
#include <dos/notify.h>
#include <dos/rdargs.h>

#include <clib/alib_protos.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

/* Use pragmas if you've got them */

static UBYTE   *VersTag = "\0$VER: MessageNotification 37.1 (09.07.91)";
struct Library *DOSBase;
struct ExecBase *SysBase;

/* I'll keep track of the NotifyRequests in an Exec list */
struct NotifyNode
{
    struct Node     nn_Node;
    struct NotifyRequest nn_NotifyRequest;
    /* and whatever is useful. Maybe a FileInfoBlock structure... */
};

VOID            main(VOID);

VOID
main(VOID)
{

    struct RDArgs  *readargs;
    LONG            rargs[2];
    struct NotifyRequest *notifyrequest;
    struct NotifyMessage *notifymsg;
    struct List    *notifylist;
    struct MsgPort *notifyport;
    struct NotifyNode *nnode, *nextnode;
    UBYTE         **filenames;
    ULONG           signal, notifysignal;
```

```
    /* To appease amiga.lib */
    SysBase = (*((struct Library **) 4));

    /* Fail silently if < 37 */
    if (DOSBase = OpenLibrary("dos.library", 37))
    {

        /* See the DOS Autodocs for more information about ReadArgs() */
        if (readargs = ReadArgs("Filename/A/M", rargs, NULL))
        {
            /* Pointer to array of filenames */
            filenames = (UBYTE **) (rargs[0]);

            if (notifyport = CreateMsgPort())
            {
                if (notifylist = AllocMem(sizeof(struct List), MEMF_CLEAR))
                {
                    /* initialize list */
                    NewList(notifylist);

                    /* The list of filenames is terminated with a NULL */
                    while (*filenames)
                    {
                        /* Get a NotifyNode */
                        if (nnode = AllocMem(sizeof(struct NotifyNode), MEMF_CLEAR))
                        {
                            /*
                             * Use ln_Name to store qualified filename to
                             * monitor. Note that I keep using that pointer to
                             * the command argument line here. In a real life
                             * application you're better off copying the
                             * string.
                             */
                            nnode->nn_Node.ln_Name = (UBYTE *) * filenames++;
                            notifyrequest = &(nnode->nn_NotifyRequest);

                            /* Initialize notifcation request */
                            notifyrequest->nr_Name = nnode->nn_Node.ln_Name;
                            notifyrequest->nr_Flags =
                                NRF_SEND_MESSAGE | NRF_WAIT_REPLY;
                            notifyrequest->nr_stuff.nr_Msg.nr_Port = notifyport;
                            /*
                             * I'm storing the address of the NotifyNode in
                             * nr_UserData. Not going to do anything with it
                             * here, but it would enable me to use the node
                             * immediately when I receive a NotifyMessage.
                             */
                            notifyrequest->nr_UserData = (ULONG) nnode;
                            /*
                             * only add the node to the list if notification is
                             * supported
                             */
                            if ((StartNotify(notifyrequest)) == DOSTRUE)
                            {
                                AddTail(notifylist, (struct Node *) nnode);
                                Printf("Notification on %s\n",
                                       (LONG) nnode->nn_Node.ln_Name);
                            }
                            else
                            {
                                Printf("Notification failed on %s\n",
                                       (LONG) nnode->nn_Node.ln_Name);
                                FreeMem(nnode, sizeof(struct NotifyNode));
                            }
                        }
                        else
                        {
                            PrintFault(ERROR_NO_FREE_STORE, NULL);
                            break;
                        }
                    }

                    /*
                     * Is list empty? If so get out of here. (Macro defined in
                     * <exec/lists.h>)
                     */
                    if (!(IsListEmpty(notifylist)))
                    {
                        /*
                         * No empty, so we've got outstanding NotifyRequests.
                         * Loop until Ctrl-C.
                         */
                        notifysignal = 1L << notifyport->mp_SigBit;
                        for (;;)
                        {
                            /* Wait for message port signals and break */
                            signal = Wait(notifysignal | SIGBREAKF_CTRL_C);

                            if (signal & notifysignal)
                            {
                                while (notifymsg =
                                       (struct NotifyMessage *) GetMsg(notifyport))
                                {
                                    /*
                                     * Here is that node again, stuffed in
                                     * nr_UserData. Can immediately reference
                                     * its data, like comparing filesize with
                                     * stored filesize in node, or remove it
                                     * (after an EndNotify() ofcourse).
                                     */
                                    Printf(
                                        "Notification message for %s, Node at 0x%lx\n",
                                        (LONG) notifymsg->nm_NReq->nr_Name,
                                        (LONG) notifymsg->nm_NReq->nr_UserData);
                                    ReplyMsg((struct Message *) notifymsg);
                                }
                            }

                            if (signal & SIGBREAKF_CTRL_C)
                            {
                                /*
                                 * Walk down the list, remove all
                                 * NotifyRequests, free all nodes.
                                 */
                                nnode = (struct NotifyNode *) notifylist->lh_Head;
                                while (nextnode =
                                       (struct NotifyNode *) nnode->nn_Node.ln_Succ)
                                {
                                    notifyrequest = &(nnode->nn_NotifyRequest);
                                    Printf("Removing notifcation for %s\n",
                                           (LONG) notifyrequest->nr_Name);
                                    /*
                                     * remove this request. EndNotify() will
                                     * also remove any messages on the message
                                     * port.
                                     */
                                    EndNotify(notifyrequest);
                                    /*
                                     * Not really needed to Remove() the node,
                                     * never going to use this list again
                                     */
                                    Remove((struct Node *) nnode);
                                    FreeMem(nnode, sizeof(struct NotifyNode));
                                    nnode = nextnode;
                                }
                                break;
                            }
                        }
                    }
                    FreeMem(notifylist, sizeof(struct List));
                }
                DeleteMsgPort(notifyport);
            }
            FreeArgs(readargs);
        }
        else
            PrintFault(IoErr(), NULL);
        CloseLibrary(DOSBase);
    }
}
```

Amiga Font Scaling and Aspect Ratio

by John Orr

[Editor's Note: This article assumes the reader is familiar with how the Amiga stores and manages bitmap fonts under pre-2.0 Releases of the operating system. For more information, see the "Graphics: Text" chapter of the RKM:Libraries and Devices Manual.*]*

The Release 2.0 OS offers a significant improvement over the Amiga's previous font resources: it now has the ability to scale fonts to new sizes and dimensions. This means, if the *diskfont.library* can't find the font size an application requests, it can create a new bitmap font by scaling the bitmap of a different size font in the same font family. The 2.04 release of the OS improved upon the *diskfont.library's* font scaling ability so the Amiga now can utilize AGFA/Compugraphic outline fonts, yielding scalable fonts that don't have the exaggerated jagged edges inherent in bitmap scaling.

The best thing about the Amiga's font scaling is that its addition to the system is completely invisible to an application program. Because the *diskfont.library* takes care of all the font scaling, any program that uses OpenDiskFont() to open a font can have scalable fonts available to it. For simple scaling, the programming interface is the same using Release 2.0 as it was under 1.3.

However, there is one feature of the 2.0 *diskfont.library* that the 1.3 programming interface cannot handle. When scaling a font (either from an outline or from another bitmap), the 2.0 *diskfont.library* can adjust the width of a font's glyphs according to an *aspect ratio* passed to OpenDiskFont(). A font glyph is the graphical representations associated with each symbol or character of a font.

The aspect ratio refers to the shape of the pixels that make up the bitmap that *diskfont.library* creates when it scales a font. This ratio is the width of a pixel to the height of the pixel (XWidth/YWidth). The *diskfont.library* uses this ratio to figure out how wide to make the font glyphs so that the look of a font's glyphs will be the same on display modes with very different aspect ratios. For more information on Amiga aspect ratios, see Carolyn Scheppner's article, "Finding the Aspect Ratio", from the September/October 1991 issue of *Amiga Mail.*

To add this new feature, several changes to the OS were necessary:

1) The OS needed to be able to store an aspect ratio for any font loaded into the system list,
2) The structures that *diskfont.library* uses to store bitmap fonts on disk had to be updated so they can store the aspect ratio a font was designed for, and
3) The way in which an application requests fonts from *diskfont.library* had to be altered so that an application could ask for a specific aspect ratio.

For the *diskfont.library* to be able to scale a font to a new aspect ratio, it needs to know what the font's current aspect ratio is. The Amiga gets the aspect ratio of a font currently in the system list from an extension to the TextFont structure called (oddly enough) TextFontExtension. Under 2.0, when the system opens a new font (and there is sufficient memory), it creates this extension.

A font's TextFont structure contains a pointer to its associated TextFontExtension. While the font is opened, the TextFont's tf_Message.mn_ReplyPort field points to a font's TextFontExtension. The *<graphics/text.h>* include file #defines tf_Message.mn_ReplyPort as tf_Extension.

The TextFontExtension structure contains only one field of interest: a pointer to a tag list associated with this font:

```
struct TagItem *tfe_Tags;/* Text Tags for the font */
```

If a font has an aspect ratio associated with it, the OS stores the aspect ratio as a tag/value pair in the tfe_Tags tag list.

The `TA_DeviceDPI` tag holds a font's aspect ratio. The data portion of the `TA_DeviceDPI` tag contains an X DPI (dots per inch) value in its upper word and a Y DPI value in its lower word. These values are unsigned words (UWORD). *At present, these values do not necessarily reflect the font's true X and Y DPI, so their specific values are not relevant. At present, only the ratio of the X aspect to the Y aspect is important* (more on this later in the article).

Notice that the X and Y DPI values are not aspect values. The X and Y aspect values are the reciprocals of the X and Y DPI values, respectively:

XDPI = 1/XAspect
YDPI = 1/YAspect

so, the aspect ratio is YDPI/XDPI, not XDPI/YDPI.

Before accessing the tag list, an application should make sure that this font has a corresponding TextFontExtension. The ExtendFont() function will return a value of `TRUE` if this font already has an extension or ExtendFont() was able to create an extension for this font.

Where Do the X and Y DPI values Come From?

The Amiga has a place to store a font's X and Y DPI values once the font is loaded into memory, but where do these X and Y values come from? A font's X and Y DPI values can come from several sources. The X and YDPI can come from a font's disk-based representation, or it can be set by the programmer.

For the traditional Amiga bitmap fonts, in order to store the X and Y DPI values in a bitmap font's *.font* file, the structures that make up the *.font* file had to be expanded slightly. There are two changes from the structures described in the "Graphics:Text" chapter of the *RKM: Libraries and Devices* manual. The first is that the fch_FileID in the FontContentsHeader structure (essentially, the *.font* file) contains a different file ID (TFCH_ID from *<libraries/diskfont.h>*) from the traditional *.font* file. This tells the *diskfont.library* that this FontContentsHeader uses a special kind of FontContents structure called a TFontContents structure. The difference between these structures is that the TFontContents structure allows the OS to store tag value pairs in the extra space set aside for the font's name.

```
struct TFontContents {
    char    tfc_FileName[MAXFONTPATH-2];
    UWORD   tfc_TagCount;/* including the TAG_DONE tag */
    /*
     *    if tfc_TagCount is non-zero, tfc_FileName is overlayed with
     *    Text Tags starting at:  (struct TagItem *)
     *        &tfc_FileName[MAXFONTPATH-(tfc_TagCount*sizeof(struct TagItem))]
     */
    UWORD   tfc_YSize;
    UBYTE   tfc_Style;
    UBYTE   tfc_Flags;
};
```

Currently, out of all the system standard bitmap fonts (ones loaded from bitmaps on disk or ROM, not scaled from a bitmap or outline), only one has a built in aspect ratio: *Topaz-9*.

For the Compugraphic outline fonts, the X and Y DPI values are built into the font outline. Because the format of the Compugraphic outline fonts is proprietary, information about their layout is available only from AGFA/Compugraphic. For most people, the format of the outline fonts is not important, as the *diskfont.library* handles converting the fonts to an Amiga-usable form.

The other place where a font can get an aspect ratio is an application. When an application opens a font with OpenDiskFont(), it can supply the `TA_DeviceDPI` tag that the *diskfont.library* uses to scale (if necessary) the font according to the aspect ratio. To do so, an application has to pass OpenDiskFont() an extended version of the TextAttr structure called the TTextAttr:

```
struct TTextAttr {
    STRPTR  tta_Name;/* name of the font */
    UWORD   tta_YSize;/* height of the font */
    UBYTE   tta_Style;/* intrinsic font style */
    UBYTE   tta_Flags;/* font preferences and flags */
    struct TagItem *tta_Tags;/* extended attributes */
};
```

The TextAttr and the TTextAttr are identical except that the tta_Tags field points to a tag list where you place your TA_DeviceDPI tag. To tell OpenDiskFont() that it has a TTextAttr structure rather than a TextAttr structure, set the FSF_TAGGED bit in the tta_Style field.

For example, to ask for *Topaz-9* scaled to an aspect ratio of 75 to 50, the code might look something like this:

```
...

#define MYXDPI (75L << 16)
#define MYYDPI (50L)

struct TTextAttr mytta = {
    "topaz.font",
    9,
    FSF_TAGGED,
    0L,
    NULL
};

struct TagItem tagitem[2];
struct TextFont *myfont;
ULONG dpivalue;
struct Library *UtilityBase, *DiskfontBase, *GfxBase;
void main(void)
{
    tagitem[0].ti_Tag = TA_DeviceDPI;
    tagitem[0].ti_Data = MYXDPI | MYYDPI;
    tagitem[1].ti_Tag = TAG_END;

...
            if (myfont = OpenDiskFont(&mytta))
             {
                dpivalue = GetTagData(TA_DeviceDPI,
                    0L,
                    ((struct TextFontExtension *)(myfont->tf_Extension))->tfe_Tags);
                if (dpivalue) printf("XDPI = %d    YDPI = %d\n",
                                ((dpivalue & 0xFFFF0000)>>16),
                                (dpivalue & 0x0000FFFF));
                /* blah, blah, render some text, blah, blah */
                ...

                CloseFont(myfont);
             }
...

}
```

Some Things to Look Out For

One misleading thing about the TA_DeviceDPI tag is that its name implies that the *diskfont.library* is going to scale the font glyphs according to an actual DPI (dots per inch) value. As far as scaling is concerned, this tag serves only as a way to specify the aspect ratio, so the actual values of the X and Y elements are not important, just the ratio of one to the other. A font glyph will look the same if the ratio is 2:1 or 200:100 as these two ratios are equal.

To makes things a little more complicated, when *diskfont.library* scales a bitmap font using an aspect ratio, the X and Y DPI values that the OS stores in the font's TextFontExtension *are identical* to the X and Y DPI values passed in the TA_DeviceDPI tag. This means the system can associate an X and Y DPI value to an open font size that is very different from the font size's actual X and Y DPI. For this reason, applications should not use these values as real DPI values. Instead, only use them to calculate a ratio.

For the Compugraphic outline fonts, things are a little different. The X and Y DPI values are built into the font outline and *reflect a true X and Y DPI*. When the *diskfont.library* creates a font from an outline, scaling it according to an application-supplied aspect ratio, *diskfont.library does not* change the YDPI setting. Instead, it calculates a new XDPI based on the font's YDPI value and the aspect ratio passed in the `TA_DeviceDPI` tag. It does this because the Amiga thinks of a font size as being a height in pixels. If an application was able to change the true Y DPI of a font, the *diskfont.library* would end up creating font sizes that were much larger or smaller than the YSize the application asked for. If an application needs to scale a font according to height as well as width, the application can adjust the value of the YSize it asks for in the TTextAttr.

As mentioned earlier, almost all of the system standard bitmap fonts *do not* have a built in aspect ratio. This means that if an application loads one of these bitmap fonts without supplying an aspect ratio, the system will not put a `TA_DeviceDPI` tag in the font's TextFontExtension: the font will not have an aspect ratio. If a font size that is already in the system font list does not have an associated X and Y DPI, the *diskfont.library* cannot create a new font of the same size with a different aspect ratio. The reason for this is the *diskfont.library* cannot tell the difference between two instances of the same font size where one has an aspect ratio and the other does not. Because *diskfont.library* cannot see this difference, when an application asks, for example, for *Topaz-8* with an aspect ratio of 2:1, OpenDiskFont() first looks through the system list to see if that font is loaded. OpenDiskFont() happens to find the ROM font *Topaz-8* in the system font list, which has no X and Y DPI. Because it cannot see the difference, *diskfont.library* thinks it has found what it was looking for, so it does not create a new *Topaz-8* with an aspect ratio of 2:1, and instead opens the *Topaz-8* with no aspect ratio.

This also causes problems for programs that do not ask for a specific aspect ratio. When an application asks for a font size without specifying an aspect ratio, OpenDiskFont() will not consider the aspect ratios of fonts in the system font list when it is looking for a matching font. If a font of the same font and style is already in the system font list, even though it may have a wildly distorted aspect ratio, OpenDiskFont() will return the font already in the system rather than creating a new one.

Due to a bug in the 2.00 through 2.04 versions of the *graphics.library* function WeighTAMatch(), the OS ignores the aspect ratio of a font when trying to determine if the TTextAttr passed to OpenDiskFont() matches a font already in the system font list. If an application asks for a font that exactly matches a font already in the font list in all ways except in aspect ratio, OpenDiskFont() will open the font in the system font list rather than creating a font with a matching aspect ratio. For example, if an application requests *CGTimes-20* with an aspect ratio of 1:2 and while scanning through the system font list OpenDiskFont() finds *CGtimes-20* with an aspect ratio of 1:1, OpenDiskFont() will open the font with a 1:1 aspect ratio. A patch that temporarily fixes this bug, *SetPatchWTAM*, is on the Denver/Milano DevCon disks.

The following example, *cliptext.c*, renders the contents of a text file to a Workbench window. This example gets the new aspect ratio for a font by asking the display database what the aspect ratio of the current display mode is.

```
;/* cliptext.c - Execute me to compile me with Lattice 5.10a
LC -cfistq -v -y -j73 cliptext.c
Blink FROM LIB:c.o,cliptext.o TO cliptext LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit

Sept. 17 1991 by John Orr
*/

#include <exec/types.h>
#include <dos/rdargs.h>
#include <dos/dosextens.h>
#include <intuition/intuition.h>
#include <graphics/text.h>
#include <graphics/displayinfo.h>
#include <graphics/regions.h>
#include <graphics/gfx.h>
#include <libraries/diskfont.h>
#include <utility/tagitem.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/layers_protos.h>
#include <clib/alib_stdio_protos.h>
#include <clib/intuition_protos.h>
#include <clib/graphics_protos.h>
#include <clib/diskfont_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE          *vers = "\0$VER: cliptext 36.5";

#define BUFSIZE         4096
#define FONT_NAME       0
#define FONT_SIZE       1
#define FILE_NAME       2
#define JAM_MODE        3
#define XASP            4
#define YASP            5
#define NUM_ARGS        6

#define DEFAULTFONTSIZE 11L
#define DEFAULTJAMMODE  0L
#define DEFAULTXASP             0L
#define DEFAULTYASP             0L

void            MainLoop(void);

LONG            args[NUM_ARGS];
struct TagItem  tagitem[2];

UBYTE           buffer[BUFSIZE];
BPTR            myfile;

struct Window  *mywin;
struct RastPort *myrp;

struct TTextAttr myta;
struct TextFont *myfont;
struct Rectangle myrectangle;

struct Region  *new_region;

struct Library *DiskfontBase, *IntuitionBase, *LayersBase, *GfxBase;
struct IntuiMessage *mymsg;
struct DrawInfo *mydrawinfo;

void
main(int argc, char **argv)
{
  struct RDArgs  *myrda;
  struct DisplayInfo mydi;
  ULONG           mymodeid;

  LONG            mydefaultfontsize = DEFAULTFONTSIZE;
  LONG            mydefaultJAMMode = DEFAULTJAMMODE;
  LONG            mydefaultXASP = 0L;
  LONG            mydefaultYASP = 0L;

  args[FONT_NAME] = (LONG) "topaz.font";
  args[FONT_SIZE] = (LONG) & mydefaultfontsize;
  args[FILE_NAME] = (LONG) "s:startup-sequence";
  args[JAM_MODE] = (LONG) & mydefaultJAMMode;
  args[XASP] = (LONG) & mydefaultXASP;
  args[YASP] = (LONG) & mydefaultYASP;

  /*
   * dos.library standard command line parsing - see the article "Standard
   * Command Line Parsing" from the MAy/June 1991 issue of Amiga Mail, or see
   * the AmigaDOS Manual for more details
   */
  if (myrda = ReadArgs("FontName,FontSize/N,FileName,Jam/N,XASP/N,YASP/N\n",
                       args, NULL))
  {
    /* open the file to display */
    if (myfile = Open((UBYTE *) args[FILE_NAME], MODE_OLDFILE))
    {
      /* open the libraries */
      if (DiskfontBase = OpenLibrary("diskfont.library", 36L))
      {
        if (IntuitionBase = OpenLibrary("intuition.library", 36L))
        {
          if (GfxBase = OpenLibrary("graphics.library", 36L))
          {
            if (LayersBase = OpenLibrary("layers.library", 36L))
            {
              /* Open that window */
              if (mywin = OpenWindowTags(NULL,
                                         WA_MinWidth, 100,
                                         WA_MinHeight, 100,
                                         WA_SmartRefresh, TRUE,
                                         WA_SizeGadget, TRUE,
                                         WA_CloseGadget, TRUE,

                /*
                 * This application wants to hear about three things: When the
                 * user clicks the window's close gadget, when the user starts to
                 * resize the window, and when the user has finished resizing the
                 * window.
                 */
                WA_IDCMP, IDCMP_CLOSEWINDOW | IDCMP_NEWSIZE | IDCMP_SIZEVERIFY,
                                         WA_DragBar, TRUE,
                                         WA_DepthGadget, TRUE,
                                         WA_Title, (ULONG) args[FILE_NAME],
                                         TAG_END))
              {
                tagitem[0].ti_Tag = TA_DeviceDPI;

                /*
                 * see if there is a non-zero value in the XASP or YASP fields.
                 * Diskfont.library will get very upset (divide by zero GURU)
                 * if you give it a zero XDPI or YDPI value.
                 */

                /* if there is a zero value in one of them... */
                if (((*(ULONG *) args[XASP]) == 0) || ((*(ULONG *) args[YASP]) == 0))
                {

                  /*
                   * ...use the aspect ratio of the current display as a
                   * default...
                   */
                  mymodeid = GetVPModeID(&(mywin->WScreen->ViewPort));
                  if (GetDisplayInfoData(NULL, (UBYTE *) & mydi,
                            sizeof(struct DisplayInfo), DTAG_DISP, mymodeid))
                  {
                    mydefaultXASP = mydi.Resolution.x;
                    mydefaultYASP = mydi.Resolution.y;

                    /*
                     * notice that the X and Y get _swapped_ to keep the look
                     * of the font glyphs the same using screens with different
                     * aspect ratios.
```

```
        */
       args[YASP] = (LONG) & mydefaultXASP;
       args[XASP] = (LONG) & mydefaultYASP;
     }
     else          /* ...unless something is preventing us from
                    * getting the screens resolution.  In that
                    * case, forget about the DPI tag. */
       tagitem[0].ti_Tag = TAG_END;
}

/*
 * Here we have to put the X and Y DPI into the TA_DeviceDPI
 * tag's data field.  THESE ARE NOT REAL X AND Y DPI VALUES FOR
 * THIS FONT OR THE DISPLAY. They only serve to supply the
 * diskfont.library with values to calculate the aspect ratio.
 * The X value gets stored in the upper word of the tag value
 * and the Y DPI gets stored in the lower word. Because
 * ReadArgs() stores the _address_ of integers it gets from the
 * command line, you have to dereference the pointer it puts
 * into the argument array, which results in some ugly casting.
 */
tagitem[0].ti_Data =
        (ULONG) (((UWORD) * ((ULONG *) args[XASP]) << 16) |
                 ((UWORD) * ((ULONG *) args[YASP])));
tagitem[1].ti_Tag = TAG_END;

/*
 * set up the TTextAttr structure to match the font the user
 * requested.
 */
myta.tta_Name = (STRPTR) args[FONT_NAME];
myta.tta_YSize = *((LONG *) args[FONT_SIZE]);
myta.tta_Style = FSF_TAGGED;
myta.tta_Flags = 0L;
myta.tta_Tags = tagitem;

/* open that font */
if (myfont = OpenDiskFont(&myta))
{

  /*
   * This is for the layers.library clipping region that gets
   * attached to the window.  This prevents the application
   * from unnecessarily rendering beyond the bounds of the
   * inner part of the window.  For now, you can ignore the
   * layers stuff if you are just interested in learning about
   * using text.  For more information on clipping regions and
   * layers, see the Layers chapter of the RKM:Libraries
   * manual.
   */
  myrectangle.MinX = mywin->BorderLeft;
  myrectangle.MinY = mywin->BorderTop;
  myrectangle.MaxX = mywin->Width - (mywin->BorderRight + 1);
  myrectangle.MaxY = mywin->Height - (mywin->BorderBottom + 1);

  /* more layers stuff */
  if (new_region = NewRegion())
  {
    /* Even more layers stuff */
    if (OrRectRegion(new_region, &myrectangle));
    {
      InstallClipRegion(mywin->WLayer, new_region);

      /*
       * obtain a pointer to the window's rastport and set up
       * some of the rastport attributes.  This example obtains
       * the text pen for the window's screen using the
       * GetScreenDrawInfo() function.
       */
      myrp = mywin->RPort;
      SetFont(myrp, myfont);
      if (mydrawinfo = GetScreenDrawInfo(mywin->WScreen))
      {
        SetAPen(myrp, mydrawinfo->dri_Pens[TEXTPEN]);
        FreeScreenDrawInfo(mywin->WScreen, mydrawinfo);
      }
      SetDrMd(myrp, (BYTE) (*((LONG *) args[JAM_MODE])));
                    MainLoop();
                  }
                  DisposeRegion(new_region);
                }
                CloseFont(myfont);
              }
              CloseWindow(mywin);
            }
            CloseLibrary(LayersBase);
          }
          CloseLibrary(GfxBase);
        }
        CloseLibrary(IntuitionBase);
      }
      CloseLibrary(DiskfontBase);
    }
    Close(myfile);
  }
  FreeArgs(myrda);
 }
 else
   VPrintf("Error parsing arguments\n", NULL);
}

void
MainLoop(void)
{
  LONG             count, actual, position;
  BOOL             aok = TRUE, waitfornewsize = FALSE;
  struct Task     *mytask;

  mytask = FindTask(NULL);
  Move(myrp, mywin->BorderLeft + 1, mywin->BorderTop + myfont->tf_YSize + 1);

  /* while there's something to read, fill the buffer */
  while (((actual = Read(myfile, buffer, BUFSIZE)) > 0) && aok)
  {
    position = 0;
    count = 0;

    while (position <= actual)
    {
      if (!(waitfornewsize))
      {
        while ( ( (buffer[count] >= myfont->tf_LoChar) &&
                  (buffer[count] <= myfont->tf_HiChar) ) &&
                (count <= actual) )
          count++;

        Text(myrp, &(buffer[position]), (count) - position);

        while ( ( (buffer[count] < myfont->tf_LoChar) ||
                  (buffer[count] > myfont->tf_HiChar) ) &&
                (count <= actual))
        {
          if (buffer[count] == 0x0A)
            Move(myrp, mywin->BorderLeft, myrp->cp_y + myfont->tf_YSize + 1);
          count++;
        }
        position = count;
      }
      else
        WaitPort(mywin->UserPort);

      while (mymsg = (struct IntuiMessage *) GetMsg(mywin->UserPort))
      {
        /* The user clicked the close gadget */
        if (mymsg->Class == IDCMP_CLOSEWINDOW)
        {
          aok = FALSE;
          position = actual + 1;
          ReplyMsg((struct Message *) mymsg);
        }
        /* The user picked up the window's sizing gadget */
        else if (mymsg->Class == IDCMP_SIZEVERIFY)
        {
```

```
        /*
         * When the user has picked up the window's sizing gadget when the
         * IDCMP_SIZEVERIFY flag is set, the application has to reply to this
         * message to tell Intuition to allow the user to move the gadget and
         * resize the window.  The reason for using this here is because of
         * the nature of the Text() routine (and most other functions that
         * use the blitter).  The Text() function only asks the blitter to
         * perform some rendering, it does not wait for the blitter to finish
         * the rendering.  Some time can lapse between the finish of the
         * Text() call and the finish of the corresponding blitter operation.
         * In this period, the user can resize the window so that the display
         * area is smaller that it was when the blitter request was made. The
         * blitter will do its rendering according to the original size of
         * the window, which can cause the blitter to write into the window's
         * borders. To prevent this, when the user attempts to resize the
         * window, we have to wait for the blitter to finish any outstanding
         * blitter operations before we allow the window to be resized.
         */

        /*
         * if this example had instead asked to hear about IDCMP events that
         * could take place between SIZEVERIFY and NEWSIZE events (especially
         * INTUITICKS), it should turn off those events here using
         * ModifyIDCMP().
         */

        /*
         * After we allow the user to resize the window, we cannot write into
         * the window until the user has finished resizing it because we need
         * the window's new size to adjust the clipping area.  Specifically,
         * we have to wait for an IDCMP_NEWSIZE message which Intuition will
         * send when the user lets go of the resize gadget.  For now, we set
         * the waitfornewsize flag to stop rendering until we get that
         * NEWSIZE message.
         */

        waitfornewsize = TRUE;
        WaitBlit();

        /* The blitter is done, let the user resize the window */
        ReplyMsg((struct Message *) mymsg);
      }
      else
      {

        ReplyMsg((struct Message *) mymsg);
        waitfornewsize = FALSE;

        /*
         * the user has resized the window, so get the new window dimensions
         * and readjust the layers clipping region accordingly.
         */
        myrectangle.MinX = mywin->BorderLeft;
        myrectangle.MinY = mywin->BorderTop;
        myrectangle.MaxX = mywin->Width - (mywin->BorderRight + 1);
        myrectangle.MaxY = mywin->Height - (mywin->BorderBottom + 1);
        InstallClipRegion(mywin->WLayer, NULL);
        ClearRegion(new_region);
        if (OrRectRegion(new_region, &myrectangle))
          InstallClipRegion(mywin->WLayer, new_region);
        else
        {
          aok = FALSE;
          position = actual + 1;
        }
      }
    }
    /* check for user break */
    if (mytask->tc_SigRecvd & SIGBREAKF_CTRL_C)
    {
      aok = FALSE;
      position = actual + 1;
    }

    /*
     * if we reached the bottom of the page, clear the rastport and move back
     * to the top
     */
    if (myrp->cp_y > (mywin->Height - (mywin->BorderBottom + 2)))
    {
      Delay(25);

      /*
       * Set the entire rastport to color zero.  This will not overwrite the
       * window borders because of the layers clipping.
       */
      SetRast(myrp, 0);
      Move(myrp,
           mywin->BorderLeft + 1,
           mywin->BorderTop + myfont->tf_YSize + 1);
    }
  }
 }
 if (actual < 0)
   VPrintf("Error while reading\n", NULL);
}
```

Keeping Time--Interval Timers in Amiga UNIX

By David Miller

The Sleep() function

When you want your program to pause for a number of seconds then continue, you will typically use the sleep(3) function (the notation NAME(SECTION#) refers to the manual page NAME in the SECTION# chapter of the *UNIX Reference Manuals*; section 1 is commands, section 2 is system calls, and section 3 is library functions). There is also a sleep(1) program which provides the same function to shell scripts. For example:

In C:

```
main()
{
    printf("Hello world!\n");
    sleep(5);
}
```

Shell:

```
#!/usr/bin/sh

echo "Hello world!"
sleep 5
```

The following example is an implementation of the sleep(3) function.

Example 1: SLEEP() Using ALARM()

```
 1  # include       <signal.h>
 2  static void trap ()
 3  {
 4  }
 5  unsigned sleep ( unsigned duration )
 6  {
 7          void            (*oldsig)();
 8          unsigned        oldtime;
 9          oldsig = signal(SIGALRM, trap);
10          oldtime = alarm(0);
11          if (oldtime && oldtime < duration)
12                  alarm(oldtime);
13          else
14                  alarm(duration);
15          pause();
16          signal(SIGALRM, oldsig);
17          if (oldtime > duration)
18                  alarm(oldtime - duration);
19          if (oldtime && oldtime < duration)
20                  return(duration - oldtime);
21          else
22                  return 0;
23  }
```

Line	Explanation
1	The file signal.h defines the parameters for the signal(2) function.
2-4	trap() is the simple signal handler that just sets a flag that another piece of code will examine.
5	Define the sleep() function.
6-8	oldsig is a variable that will be used to save the previous state of the signal handler; oldtime will be used to save the state of the alarm clock.
9	Establish trap() as the current signal handler for the alarm signal. The previous handler is saved in oldsig.
10	Clear the alarm clock, saving the current state.
11-14	If the previous setting of alarm was sooner than duration, use the old value, otherwise use duration to set the alarm.
15	Pause the process until the alarm goes off.
16	Restore the old signal handler.

17-18 If the old alarm setting was later than duration, reset the alarm with the difference between duration and oldtime (the time remaining until the previous alarm).

19-22 If an existing alarm is making sleep return early, return the time remaining on the requested sleep.

To schedule its "wake up" time, sleep(3) uses the alarm(2) system call. The alarm(2) system call asks the OS to deliver a signal (basically a software interrupt) in some number of seconds according to the system clock. However, setting an alarm for 2 seconds does not mean that you will receive an alarm signal in exactly two seconds.

The OS processes alarm requests once every second. Each time an alarm request is processed, the number of seconds remaining for that alarm is decremented by one. When the number of seconds remaining reaches zero, the OS delivers a signal to the process. Given this, if a process places a 1 second alarm request 1 microsecond before the OS does its alarm processing, the signal will arrive in 1 microsecond, not in 1 second. An alarm set for N seconds actually means:

deliver a signal after N-1 seconds, but before N seconds.

If you'd like to see for yourself, try running this shell script:

```
$ for i in 1 2 3 4 5 6 7 8 9 10
> do
>  time sleep 2
> done
```

How many times did the process actually take more than 2 seconds? If you try it with both the Korn shell, ksh(1), and the System V shell, sh(1), you'll find that under ksh(1) it takes about 1.2 seconds and under sh(1) it takes about 1.8 seconds (There is a difference because sh(1) counts the time required to start the sleep process whereas ksh(1) only counts the actual running time). The sleep(1) program rarely, if ever, actually sleeps for 2 seconds.

If you are writing a daemon that checks for some event every 5 minutes, or if you want to pause the output to give the user a chance to read it, alarm's 1 second granularity is fine. But what about that daemon that needs to wake up every second? Waking up after 1 microsecond could cause the process to run almost continuously. For any sort of realtime processing, one second is a very long time. So how do you sleep for less than one second reliably?

The answer is do not use an alarm, use an *interval timer*.

Interval Timers

Each interval timer has a resolution of 1 tick of the system's clock, or 1 microsecond (whichever is larger). Additionally, you can configure an interval timer to automatically restart itself. The system provides each process with three independent interval timers:

ITIMER_REAL This timer will count down in real time. That is, this timer will continue to run when your process is waiting for the OS to perform a system call, or when the OS preempts your process. When the timer expires, the OS will deliver a SIGALARM signal.

ITIMER_VIRTUAL This timer counts down only when your process is running. If your process makes a system call, or is preempted, this timer will stop counting. The timer will resume when your process resumes execution. When this timer reaches zero, the process will receive a SIGVTALRM signal.

Possible uses for this timer include checkpointing (saving data after some period of execution) and multithreading. The virtual timer is more desireable for these applications since it counts only when the process is running; there is no reason to perform a checkpoint or switch threads if the process has been idle.

ITIMER_PROF This timer will stop counting any time your process is preempted by the OS, but will not stop when the process is waiting for a system call to return. When it expires the OS generates a SIGPROF signal.

This timer is designed to be used for execution profiling by interpreters. By having a profiling timer send a signal every second, or fraction of a second, and examining the current position in the interpreted code, the process can determine where the most execution time is being spent.

All three timers operate on the following structure:

```
struct itimerval
{
        struct timeval it_interval;
        struct timeval it_value;
}
```

The timeval structure looks like this:

```
struct timeval
{
        long tv_sec;
        long tv_usec;
}
```

Both of these structures are defined in the *<sys/time.h>* include file.

Note that System V Release 4.0 does not guarantee that these are the only members of these structures, nor that they will occur in this order. You must initialize the members individually. This can be annoying and tedious, but it allows the structure to be expanded in future releases.

You set and examine timers using these two functions:

```
int getitimer(int which, struct itimerval *myvalue)
```

places the current timer setting into myvalue

```
int setitimer(int which, struct itimerval *myvalue, struct itimerval *myovalue);
```

sets the timer. It gets the new time from myvalue and places the previous setting in myovalue.

Now, let us take a look at the sleep function again. The code in example 2 creates a new version of sleep that will sleep an exact number of seconds.

Example 2: SLEEP() Using an Interval Timer

```
 1  # include       <signal.h>
 2  # include       <sys/time.h>
 3  static void trap ()
 4  {
 5  }
 6  unsigned mysleep ( unsigned duration )
 7  {
 8          void                    (*oldsig)();
 9          struct itimerval        oldtime;
10          struct itimerval        newtime;
11          oldsig = signal(SIGALRM, trap);
12          getitimer(ITIMER_REAL, &oldtime);
13          if (oldtime.it_value.tv_sec == 0
                    && oldtime.it_value.tv_usec == 0
                    || oldtime.it_value.tv_sec >= duration)
14          {
15                  newtime.it_interval.tv_sec = 0;
16                  newtime.it_interval.tv_usec = 0;
17                  newtime.it_value.tv_sec = duration;
18                  newtime.it_value.tv_usec = 0;
19                  setitimer(ITIMER_REAL, &newtime, NULL);
20          }
21          pause();
22          signal(SIGALRM, oldsig);
23          if (oldtime.it_value.tv_sec > duration)
24          {
25                  oldtime.it_value.tv_sec -= duration;
26                  setitimer(ITIMER_REAL, &newtime, NULL);
27          }
28          if (oldtime && oldtime < n)
29                  return(n - oldtime);
30          else
31                  return 0;
32  }
```

Line	Explanation
1-2	The file *signal.h* defines the parameters of the signal(2) function. The file <*sys/time.h*> defines `ITIMER_REAL` and the structures used by getitimer(3) and setitimer(3).
3-5	trap() is the simple signal handler that just sets a flag that another piece of code will examine.
6-7	Define the mysleep() function.
8-10	oldsig will hold the previous state of the signal handler; oldtime will hold the state of the timer; and newtime will be used to set the new timer parameters.
11	Establish trap() as the current signal handler for the alarm signal. The previous handler is saved in oldsig.
12	Fetch the current settings of the timer.
13-20	If the timer was idle (both parts of it_value are zero) or it is set to go off later than duration (it_value.tv_sec is greater than duration), set the timer to go off in duration seconds.
21	pause(2) the process until the timer expires.
22	Restore the old signal handler.
23-27	If the old timer setting was later than duration, reset the timer with the difference between duration and oldtime.it_value.tv_sec (the time remaining until expiration of the previous setting).
28-32	If an existing timer is making mysleep() return early, return the time remaining on the requested mysleep().

This example uses what is called a "one-shot" timer. The timer goes off once, and then stops. By supplying an interval setting, the timer becomes a clock, generating alarm signals on a regular basis.

The code fragment in Example 3 shows how to set the timer to produce a 1.5 second clock that will start "ticking" in 1 minute.

Example 3: Using an Interval Timer as a Clock

```
1   # include        <sys/time.h>
2   ...
3           {
4                   struct itimerval        newtime;
5                   newtime.it_value.tv_sec = 60;
6                   newtime.it_value.tv_usec = 0;
7                   newtime.it_interval.tv_sec = 5;
8                   newtime.it_interval.tv_usec = 500000;
9                   setitimer(ITIMER_REAL, &newtime, 0);
10          }
11  ...
```

Line	Explanation
1	The file *<sys/time.h>* defines **ITIMER_REAL** and the structures used by getitimer(3) and setitimer(3).
2	Other code.
3	Start of block.
4	Structure newtime will hold the setting for the timer.
5-6	Set it_value to expire in 1 minute.
7-8	Set it_interval to reload it_value with 5 seconds and 500,000. microseconds (one half of a second).
9	Load the new settings into the **ITIMER_REAL** timer.
10	End of block.
11	Other code.

When the time interval specified by it_value expires, the contents of it_interval is copied into it_value and the timer is restarted. If the interval specified by it_interval is zero, the timer stops. Timers can be stopped at anytime by calling setitimer(3) with the members of it_value set to zero.

So there you have the realtime interval timer. The other timers work exactly the same way, varying only in when the timer is running. For more information on interval timers see *UNIX System V Release 4 - Programmer's Reference Manual*, published by Prentice Hall. Next time, we'll explore context switching, multithreading, and light weight processes.

Amiga Mail
Volume II
November/December 1991

From the Desk of Paul Higginbottom

It's time for all of us to work together more efficiently and effectively. We all know that Amiga systems and the available hardware and software are exciting, powerful, creative, and a great value for the money. However, they could be so much more exciting, powerful, creative and an even better value if there weren't so much duplication of effort by developers and if products worked more effectively together.

This does not mean that I am opposed to competition in the Amiga market. Compiler wars, spreadsheet wars, editor wars, and so on, have meant the continuing release of better and better products. But unless a new product is significantly more powerful, cheaper, or different from other available products, it is unlikely to succeed in the market.

If we all work together, the Amiga will be a stronger competitor in the marketplace. You may be thinking that this is something that only Commodore should worry about, but the Amiga's competitiveness directly affects and is affected by the size and growth of the market that you sell into.

Working together more effectively means producing products that work well with other complementary products on the market. Due to a lack of user interface and file interchange standards on the PC, software vendors created so-called "integrated" packages. Integrated packages consist of a number of applications such as a word processor, spreadsheet, and database that each utilize a common user interface, and can exchange data. These huge packages provide the user with a simple, seamless environment, but one that is basically constrained to the applications contained in the integrated package.

If applications from different vendors do not have a consistent look and feel, a consistent, but huge, integrated program may look attractive to the user. But, if applications from different vendors are consistent, and work well together, the platform itself is much more attractive, giving the user the ability to find the tools that match their needs without worrying about learning massively different user interfaces, and dealing with incompatibilities. That is why conforming to standards to make your products consistent from the users' point of view is so important. We don't want or need huge integrated packages. This is the Amiga we're talking about. The machine that can run multiple packages at once. The machine that offers operating system and user level communication between programs (Exec and ARexx respectively). So design your products to not only work well by themselves, but to work well with other products also.

If you cannot beat a product in an existing category in some way (price, performance, unique features, etc.), you should focus on developing products where there are holes in the market. Such a product may not be a new concept, but simply a bridge between existing applications that don't work well together.

Another way for us to work more efficiently together is to sell your strengths, and buy your weaknesses. If you are developing a product but lack the expertise in one area of development, buy it from another developer. There is tremendous talent in the developer community, but there is also too strong a desire to do everything alone. If you know that you have an area of development expertise, market and sell it to other developers! We (CATS) will gladly distribute flyers from you to other developers touting your available expertise. All you need to do is contact myself or Susan West in CATS about this, and we can advise you on what to do. Usually we ask you to send us a box of flyers, and we take care of the rest.

At the recent DevCons, I was very impressed by the increasing level of professionalism in the Amiga developer community. In order to be more successful, we now need to focus our efforts, capitalize on our strengths, and work on our weaknesses.

Remember, if you have any questions, suggestions, or criticism, feel free to give me a call. My number is (215)431-9228. You can send email to me care of someone in CATS on BIX if you wish, or through uucp at "higgin@cbmvax.commodore.com" or "{rutgers| uunet}!cbmvax!higgin".

-PH

Table of Contents

November/December 1991

I
Programming Practices and Standards
general purpose Amiga programming

Tag Items and Tag Lists . . . . . I - 1
staff

2.0 Version Strings . . . . . I - 3
Carolyn Scheppner

The ASL Library. . . . . . I - 7
John Orr

2.0 Compatibility Problem Areas . . . . . I - 19
Carolyn Scheppner, Bryce Nesbitt, Peter Cherna, and Darren Greenwald

II
AmigaDOS
dos.library

Executing External Programs with Release 2.0 . . . . . II - 1
Carolyn Scheppner

AmigaDOS Packet Interface Specification . . . . . II - 5
John Toebes

Standard Command Line Parsing . . . . . II - 27
John Orr

Notification . . . . . II - 31
Ewout Walraven

Using the AmigaDOS Pattern Matching Functions . . . . . II - 37
Ewout Walraven

The 2.0 Dos.library Path Name Handling Functions . . . . . II - 43
Ewout Walraven

Directory Scanning . . . . . II - 49
Ewout Walraven

III
Exec
the multitasking executive, exec.library

Using SetFunction() in a Debugger . . . . . III - 1
Ewout Walraven

68040 Compatibility Warning . . . . . III - 11
Michael Sinz

IV
Intuition and Workbench
the Amiga user interface

Obtaining an Icon Image . . . . . IV - 1
David Junod

An Introduction to V36 Screens and Windows . . . . . IV - 3
Ewout Walraven

Opening Screens and Windows on Any Amiga . . . . . IV - 17
Ewout Walraven

Introduction to Boopsi . . . . . IV - 23
John Orr

Writing a Boopsi Image Class . . . . . IV - 41
David N. Junod

Boopsi's Transparent Base Classes . . . . . IV - 57
Peter Cherna

AppWindows, AppIcons, and AppMenuItems . . . . . IV - 59
Fred Mitchell and John Orr

Making Your Windows More Shiny and Manageable . . . . . IV - 69
Ewout Walraven

V
Graphics
animation, text, the Blitter, layers.library

Using the ECS Genlocking Features . . . . . V - 1
Ewout Walraven

Finding the Aspect Ratio . . . . . V - 11
Carolyn Scheppner

Amiga Font Scaling and Aspect Ratio . . . . . V - 15
John Orr

VII
IFF
iffparse.library and other IFF topics

The DR2D FORM . . . . . VII - 1
Ross Cunniff and John Orr

ECS Display Modes and ILBM CAMG . . . . . VII - 17
Carolyn Scheppner

Developing New IFF FORMs and Chunks . . . . . VII - 21
Cristian Ludwig

VIII
Unix and Networking
Unix SVR4, NFS, TCP/IP, Ethernet

Using the Amiga A3000UX as a NFS File Server . . . . . VIII - 1
David Miller

Keeping Time--Interval Timing in Amiga UNIX . . . . . VIII - 9
David Miller

XI
Input
user input, Commodities Exchange

Introduction to Commodities Exchange . . . . . XI - 1
John Orr

Commodore
International Services Corporation

1200 Wilson Drive
West Chester, PA 19380
(215) 431-9100

Technical Information List for Non-Registered Developers and Users

Listed below are the materials available through Department C of Commodore Business Machines, Inc., 1200 Wilson Drive, West Chester, PA, 19380. Send check or money order only (no credit card orders) to Department C at the above address. These materials are shipped from the new warehouse location in Mechanicsburg, PA.

Shipping and handling for USA is $3.00. Shipping and handling for Canada and Europe (via Air) is from $5.00 to $15.00 depending on weight of shipment.

Add local state sales tax for the following states: Pennsylvania, Virginia, California, Tennessee and Illinois.

Allow 4-6 weeks for delivery.

A500/A2000 Technical Reference Guide, P/N: TECHREF01 $40.00

A 275-page reference manual that describes the technical features of the A500 and A2000, as well as those features that differ from the A1000. Table of contents includes: System Block Diagrams, Amiga Expansion, Designing Hardware for the Amiga Expansion Architecture, Driver Documentation, Software for Amiga Expansion, PC Bridgeboard and schematics.

Software Toolkit, P/N:SWT01 $25.00

(1988) Manual and disk set which includes several utilities to assist developers with Amiga Programming: WACK, Memacs, Debug, Keytoy, etc.

IFF Manual and Disk, P/N:IFF01 $20.00

(October, 1988) Spiral-bound manual includes full IFF documentation and source listings for Amiga IFF. The IFF disk contains source code, object files, executable programs and documentation.

Programmer's Guide to ARexx and Disk, P/N: AREXX01 $20.00

(August 1991) Manual (228 pages) and disk designed to allow you easy access to the power of ARexx. Includes information on how to make effective use of ARexx, how to conform to development standards and how to interface applicatations to the ARexx environment.

1990 Amiga Developers Conference Notes, P/N:NOTES90 $125.00

850-page manual and 2 disks covering the 1990 Amiga Developers Conference. Included are sections on Amiga OS 2.0, A3000 Local Bus Expansion Connector, Zorro III Bus Specification, A3000 Video Board Form Factor, Debugging Tools, GadTools, Programming SCSI, and Implementing ARexx.

1989 Amiga Developers Conference Notes, P/N:NOTES89 $75.00

510-page manual and 2 disks created for 1989 Amiga Developers Conference. Subjects include: Intro to Programming the Ammiga, Features Outline for V1.3 ECS Features and the Graphics Library, Janus Dual-Port Memory, Hi-Res Color Graphics Card, Interfacing to ARexx, Advanced Amiga Architechtures, The IFF parse.library, and more.

1988 Amiga Developers Conference Notes, P/N:NOTES88 $75.00

400+ page manual and 4 disks created for 1988 Amiga Developers Conference. Topics covered include: Unique Amiga Techniques, Tips and Tricks for Programming in C, IFF, Autoboot and Kickstart V1.3, V1.3 Printer Device and Printer Drivers, Amiga Audio and Sound, Overscan, Hi-Resolution Fonts, Programming for 16-Bit Amiga, A500 Expansion Cards, and much more.

AmigaDOS V1.3 Native Developer Update, P/N:NATDEV01 $20.00

(November 1988) A four-disk set for the native Amiga developer. The following files are contained on the disk: Include 1.3; Commented C and Assembler Include files, include.strip 1.3; stripped include files, 1.3 startups, Alink, Amiga.lib, debug.lib ddebug.lib, atom, LVO offset lists, and more. Autodocs 1.3; 1.3 autodocks summarize library, resource, and device calls, Readme 1.3, Readme files and sample 2.3 pointer driver sources.

Amiga 1000 Schematics and Expansion Specifications, P/N:A1000SM $20.00

(June 1986) Spiral-bound manual containing full Amiga 1000 schematics, timing diagrams, PA: equations, and documentation for the auto-configuration process.

AmigaMail Volume 1, P/N:AMVOL1 $75.00

Back issues of the technical newsletter on the Amiga from January/February 1987 to July/August '90. (Binder and index tabs are sold separately.)

AmigaMail Binder, P/N:AMAIL1B01 (Vol. 1), AMAIL2B02 (Vol. 2) $10.00 ea.

AmigaMail Index Tabs, P/N:AMAIL1IND01 (Vol. 1) AMAIL2IND02 (Vol. 2) $5.00 ea.

The following item may be ordered by non-registered developers or users; however, it must be ordered directly from the CATS Department at Commodore. IF YOU ARE ORDERING THE FOLLOWING ITEM, DO NOT COMBINE THIS ORDER WITH THE DEPARTMENT C MATERIALS LISTED ABOVE.

AmigaMail, P/N:AM101 $45.00 USA Annual Subscription

45.00 + $2.50 (shipping) - (Canada)

45.00 + $15.00 (shipping) - European Orders Using Credit Card Visa and MasterCard Only

45.00 + $25.00 (processing fee) + $15.00 (shipping) on European orders paid with check or money order

Bi-monthly technical publication.